UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de un interfaz gráfico basado en EMF para el modelado de Petri net Product Lines

Víctor García-Bermejo Mazorra
Tutor: María Elena Gómez Martínez
Ponente: Juan de Lara Jaramillo

Julio 2020

Desarrollo de un interfaz gráfico basado en EMF para el modelado de Petri net Product Lines

AUTOR: Víctor García-Bermejo Mazorra
TUTOR: María Elena Gómez Martínez

Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Julio de 2020

Resumen

Un sistema concurrente es aquel que tiene varios elementos que pueden hacer su trabajo en paralelo. La importancia de formar correctamente uno de estos sistemas, reside en la cantidad de tiempo y recursos que se puede optimizar. Una red de Petri es una forma de modelar y analizar sistemas concurrentes. Sin embargo, la representación de variantes de un mismo sistema concurrente, reduciendo o aumentando este, no se puede realizar mediante una única red de Petri estándar. Como solución es posible incluir flujos alternativos (Líneas de Producto) a redes de Petri cuya activación permite representar múltiples redes en un mismo modelo.

Una Línea de Producto es un conjunto de productos que tienen unas propiedades comunes pero que difieren en ciertos aspectos particulares. Estos últimos se denominan *feature* y si afectan a una propiedad del producto concreto, cada *feature* puede ser tratada como una variable lógica. Llevando al terreno de las redes de Petri estas definiciones, una *Petri net Product Line* (PNPL) es una línea de productos de redes de Petri. En la actualidad, no existen herramientas que permitan el modelado grafico de PNPL.

Con el fin de solventar este problema, el objetivo principal de este Trabajo de Fin de grado es crear una herramienta de modelado de PNPL que funcione como un complemento del Entorno de Desarrollo Eclipse. Para ello, se usaron la generación de archivos basada en proyectos de modelado que proporciona Eclipse Modeling Framework (EMF) y Sirius, un complemento de Eclipse que sirve para generar herramientas de diseño. Además, tras la finalización de un modelo con la herramienta, se deberá obtener una serie de ficheros que serán usados para el análisis de PNPL y que deberán cumplir el estándar de compatibilidad entre aplicaciones de redes de Petri llamado *Petri Net Markup Language* (PNML).

A lo largo de este documento, se detalla el desarrollo paso a paso de todos los elementos que conforman los dos objetivos. Inicialmente se ofrecen una serie de motivaciones y conocimientos básicos asociados al proyecto. A continuación, se hace un breve recorrido del estado del arte y después se procede con el diseño y desarrollo de las tareas. Finalmente se detallan de forma breve las pruebas que se han ido haciendo a lo largo del desarrollo y unas conclusiones.

Palabras clave

Red de Petri, variabilidad, Eclipse, metamodelo, herramienta de diseño, complemento de Eclipse, línea de producto de red de Petri, ecore.

Abstract

Parallel systems are those systems that have multiple elements that can perform parallel tasks. The time and resources used to optimize these systems are key in order to correctly assemble these systems. A Petri Net is a method to model and analyse these systems. However, the issue of representing different alternatives of a given model, increasing or decreasing its functionality in the process, cannot be done using a single standard Petri net. Instead, it is possible to include alternative branches (Product lines) to Petri nets, which allow to multiple net representations on the same model.

A Product Line, is a set of products that share some common properties but also have certain particular aspects on their own. These last aspects are called /features/ and if they refer to any property of a product, they are treated as logical variables. Bringing this concept to the Petri nets' area of knowledge, a Petri net Product Line (PNPL) is a line of products of Petri nets. Nowadays, there is no tool that can graphically represent models of PNPLs.

With the aim of solving this issue, the main objective of this work is to create a PNPL modelling tool that can work as a complement of the Develop Environment /Eclipse/. Thus, Eclipse Modeling Framework (EMF) and Sirius (Eclipse plugin), were used to generate: files based on modelling projects and design tools, respectively. In the same vein, and after a model is finished using these tools, a set of files will be obtained, in order to being used to the posterior analysis of the PNPL, that will have to follow the so-called Petri Net Markup Language (PNML) as a compatibility standard between Petri Nets.

Along this document, the step by step process of the development, of every element, is detailed following the previously defined objectives. First a series of motivations and basic knowledge associated to the area of study are provided. Next, a brief summary of the state of the art is presented, followed by the design and development of the different tasks. Finally, the test performed over the duration of this project are detailed along with the conclusions.

Keywords

Petri net, variability, Eclipse, meta-model, Eclipse’s plug-in, Petri net’s product line, ecore.

Agradecimientos

A mi tutora Elena por haber sido capaz de soportarme y ayudarme en todo momento. Se agradece que haya personas tan dedicadas y tranquilas como tú.

A mis padres y mi hermano, por haberme apoyado y haber tirado de mí para hacer esta carrera. Sin vosotros hubiera sido más difícil encontrar algo que me gustase tanto.

A mis amigos por haber estado ahí, tanto en los buenos como en los malos momentos. Es anecdótico porque yo siempre me he ido moviendo de grupo en grupo y me hace feliz saber que de todos vosotros he sacado cosas buenas. Por ello, este trabajo está dedicado a todos vosotros.

Finalmente, a todas las personas que me han acompañado durante estos años de viaje. Todos y cada uno de vosotros habéis puesto de alguna forma, algo de sal a esta sosa historia. Siempre fue un placer compartir momentos con todos.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

Un sistema concurrente es aquel que tiene varios elementos que pueden hacer su trabajo en paralelo. El uso de estos sistemas abarca un gran marco, desde el manejo de procesos en la informática hasta modelos de negocio. La importancia de formar correctamente uno de estos sistemas, reside en el tiempo y los recursos que se pueden optimizar. Mediante la generación de una red de Petri, podemos representar y analizar cualquier sistema concurrente. Para este tipo de modelado, existe una gran cantidad de aplicaciones para trabajar redes de Petri.

Los programas que se utilizan para el diseño de redes de Petri, normalmente te permiten hacer un diseño valido para un sistema a representar. El problema de esto es que si se quieren representar varias redes de Petri que sean semejantes pero que difieran en ciertos elementos que las componen, habría que hacer nuevos diseños tantas veces como redes de Petri se quieran representar.

Este grupo de redes de Petri mencionado anteriormente, correspondería a la definición de *Petri Net Product Line* (PNPL). Una Línea de Productos es un conjunto de productos que tienen una gran cantidad de propiedades comunes, pero que sin embargo difieren en ciertos aspectos particulares. A estos últimos, se les da el nombre de *feature* y pueden ser tratadas como variables lógicas en función de si afectan a un producto concreto o no. Aplicando estos conceptos a redes de Petri, una PNPL es una Línea de Productos de redes de Petri, donde el conjunto de las redes que la forman tienen una parte común y partes específicas para cada red dentro de la Línea de Productos. Actualmente, no hay ninguna aplicación que permita el modelado gráfico de PNPL.

1.2 Objetivos

El objetivo principal de este Trabajo de Fin de Grado es crear una herramienta de modelado de PNPL ya que, a pesar de la gran variedad de aplicaciones para el modelado de redes de Petri que existen, no hay ninguna que trabaje las PNPL a raíz de la variabilidad de sus *features*. Esta herramienta de modelado, se deberá implementar para que funcione a modo de complemento o *plug-in* para el Entorno de Desarrollo Eclipse que nos dará como salida una serie de ficheros basado en el diseño de PNPL, para su posterior análisis mediante una serie de algoritmos.

Para ello, se utilizarán la generación de archivos basada en proyectos de modelado que proporciona Eclipse Modeling Framework utilizando una serie de metamodelos dados, y Sirius, un complemento de Eclipse que sirve para generar herramientas de diseño. Adicionalmente se debe crear un sistema de procesado de archivos con el diseño para obtener los ficheros salientes esperados, los cuales deberán cumplir el estándar *Petri Net Markup Language* (PNML) para que tengan compatibilidad con otras aplicaciones.

1.3 Organización de la memoria

La memoria del presente Trabajo Fin de Grado consta de los siguientes capítulos:

En el Capítulo 2, se hará un repaso teórico a diversos tipos de diagramas de representación de sistemas básicos, para finalmente explicar de forma sencilla una serie de conceptos básicos de las redes de Petri.

El Capítulo 3 describirá varias aplicaciones y sus principales características de redes de Petri, hasta encontrar dónde se sitúa nuestra herramienta de modelado. También se analizarán de forma simple las tecnologías usadas en el proyecto.

El Capítulo 4 planteará y asentará las bases de cómo hacer la herramienta de diseño acorde a los metamodelos dados. Inicialmente se propondrá una unión de los metamodelos, y se finalizará proponiendo dos módulos. Uno que sea el diseñador y otro que sea un procesador de archivos.

El Capítulo 5 será una sección detallada de forma cronológica de la implementación de cada módulo del proyecto.

En el Capítulo 6, se repasarán las diversas pruebas que se han hecho a todos los módulos del proyecto de forma breve. Incluye la generación de esquemas complejos y pruebas a las funciones más complejas.

En el Capítulo 7, se hace una reflexión acerca de lo aprendido en la realización del proyecto añadiendo mejoras y posibles implementaciones futuras.

2 Marco teórico

Como la aplicación que se pretende hacer en Eclipse es una interfaz de diseño gráfico basado para líneas de producto de redes de Petri (*Petri Net Product Lines,* PNPL), hay una serie de conceptos básicos que hay que tener en cuenta.

2.1 Redes de Petri

Una red de Petri es un grafo orientado en el que intervienen dos clases de nodos, los lugares (representados por circunferencias) y las transiciones (representadas por segmentos rectilíneos) unidas alternativamente por arcos [1].

Centrándonos en los elementos mencionados en la definición, es conveniente explicar su objetivo:

- Los lugares tratan de representar una fase donde se acumulan elementos en un estado de la red de Petri.
- Las transiciones tratan de aplicar una función a los elementos de un lugar dando su resultado en otro lugar o en el mismo en función de los arcos entrantes y salientes de la misma transición.

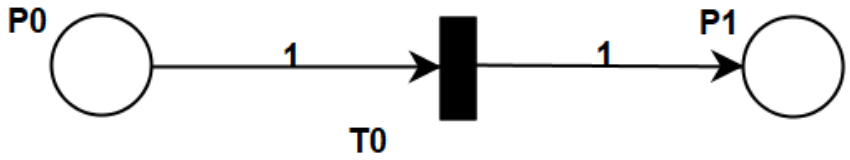

Figura 2-1. Red de Petri sencilla, acorde a la definición dada.

Es importante añadir a la definición anterior una serie de restricciones que se deben tener en cuenta a la hora de definir cualquier red de Petri:

- La primera trata de que un lugar no puede estar unido a otro lugar mediante un simple arco. Dos lugares solo pueden estar conectados a otros lugares, si entre ellos existe una transición a la cual ambos están unidos mediante arcos. Esto mismo aplica a transiciones.
- La segunda trata de la importancia de un elemento no señalado en la definición anterior, los *tokens* o marcas. Los *tokens* sirven para representar el estado actual del sistema concurrente que simboliza una red de Petri y se representarían dentro del grafo como puntos en el interior de cualquier lugar.

La importancia de estas dos propiedades reside en que, una red de Petri pretende dar una representación de una situación exacta de un sistema, que se podría entender como un estado concreto, el cual se encuentra distribuido [2].

La manera más sencilla para entender que se pretende con la representación de un estado, consiste en la situación de los *tokens*. Una representación concreta de una red de Petri con *tokens* situados de una forma arbitraria en los distintos lugares de la misma es diferente de

otra con los mismos lugares y transiciones, pero con distinta situación de los *tokens*. La situación concreta de todos los *tokens* dentro de una red de Petri, nos dan un estado del sistema que se representa.

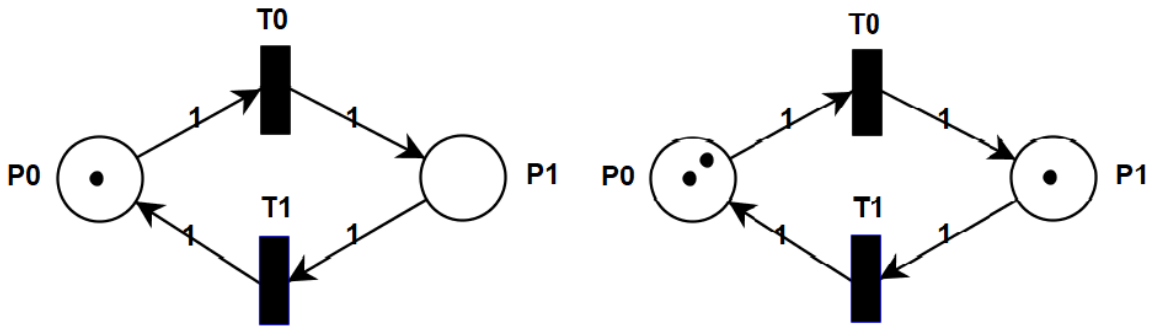

Figura 2-2. Redes de Petri que representan un mismo sistema, pero en distintos estados.

La Figura 2-2 es un ejemplo de lo explicado anteriormente. Ambas redes de Petri están representando el mismo sistema; sin embargo, debido a la situación global de los *tokens*, la representación no se encuentra en el mismo estado. Como además, dentro de un lugar, si no hay ninguna restricción que lo prohíba, puede haber un número ilimitado de *tokens*, podemos concluir, que podemos tener un número ilimitado de estados.

2.2 Líneas de Producto en redes de Petri

Una de las características más importantes de las redes de Petri, es la gran capacidad adaptativa que tienen, pudiendo ajustarse a la necesidad del sistema que quieren representar. Si bien es cierto que, muchas de ellas pueden ser semejantes pero diferir en algunos aspectos muy concretos.

Una Línea de Producto es un conjunto de productos que tienen una gran cantidad de propiedades que son comunes, pero que, en ciertos aspectos difieren. Estos aspectos en los que difieren se llaman características o *features*. Estas *features*, el hecho de que formen o no parte de un producto de la Línea de Producto se puede considerar como variables booleanas. Esto se denomina condición de presencia o *presence condition*. El conjunto de todos los valores asignados a las *features* mediante sus *presence conditions*, definen un producto concreto dentro de la Línea de Productos [3] [4] [5].

Además, el conjunto de todas las *features* que definen cualquier elemento dentro de una Línea de Producto se denomina *feature model*. Usando una definición más ajustada, un *feature model* consiste en un conjunto de *features* y una formula proposicional que fija las configuraciones concretas que tienen las *features* permitidas.

Para entender la formación de una línea de producto de una red de Petri, hay que entender que dicha red está compuesta por un *feature model*, una misma red de Petri (la cual suele ser comúnmente llamada *150% Petri net*) y un conjunto de funciones que mapean sobre los diversos elementos de la red de Petri (lugares, arcos o transiciones) las *features* que correspondan a cada uno de ellos. Entonces se entiende que, en función de los valores lógicos

que tengan las *presence conditions* asociadas a los elementos de la red de Petri, se pueden generar redes de Petri distintas.

El objetivo de nuestra aplicación es intentar montar una herramienta de modelado de PNPL, que nos dé la posibilidad, de ser utilizada para representar distintas redes de Petri a la vez, simplemente cambiando su *features*.

3 Estado del Arte

A lo largo de este capítulo, se explicarán tecnologías cercanas a la nuestra, de tal forma que se pueda ver donde se sitúa nuestra aplicación, y las herramientas utilizadas para su desarrollo.

3.1 Herramientas para redes de Petri

En la actualidad existen numerosas aplicaciones que pretenden ser utilizadas para el modelado de redes de Petri. Hay un matiz que hay que tener en cuenta cuando buscamos aplicaciones de este tipo. No es lo mismo una aplicación con la cual se puede diseñar una red de Petri genérica, que otra aplicación que pretende diseñar una línea de producto de redes de Petri. En el primero de los casos el modelado atiende una relación 1-1 respecto con los sistemas que quiere representar, mientras que el diseño de una PNPL tener una relación 1-N siendo N el número de redes de Petri que contiene la Línea de Producto. Independientemente de esto, existe una base de datos pública donde están almacenadas todas las aplicaciones relacionadas con redes de Petri [6]. Dentro de las 97 aplicaciones registradas en la base de datos, la mayor parte de ellas no son mantenidas actualmente por sus creadores. A continuación nos centraremos en las aplicaciones que si han sido mantenidas y, además, han seguido los estándares de creación de herramientas de diseño de redes de Petri marcados por el *Petri Net Markup Language* (PNML) [7].

3.1.1 Platform Independent Petri Net Editor 2 (PIPE)

Es una aplicación de código libre desarrollada en Java por un grupo de alumnos de máster del *Imperial College London* (Reino Unido) entorno al año 2007 con el objetivo de que un usuario pudiese utilizarla de manera sencilla, simplemente sabiendo cómo funcionan las redes de Petri. Por ello, tiene una interfaz bastante amigable. Dado que sigue el estándar PNML, tiene una gran compatibilidad con otras herramientas pudiendo importar y exportar fichero de redes de Petri creados en otras aplicaciones.

Dentro de sus características más relevantes, encontramos que esta herramienta nos permite simular los sucesos marcados en las transiciones de la red de Petri que se han creado. Con ello, en vez de tener una red de Petri estática, que solo represente un estado de ésta, tendremos una red dinámica donde podemos simular los distintos estados que se forman mediante el movimiento de *tokens* usando las transiciones [8] [9].

3.1.2 GreatSPN

Esta aplicación es una herramienta de modelado perteneciente al departamento de informática de la Universidad de Turín (Italia). El objetivo principal es proveer a los usuarios de la capacidad de modelar redes de Petri generalizadas estocásticas. La idea consiste en dotar tiempo a los estados de una red de Petri en función de sus transiciones.

Al igual que la mayor parte de las herramientas de la base de datos, además de un formato nativo, GreatSPN nos permite leer y utilizar archivos de otras aplicaciones almacenadas en la base de datos, puesto que siguen el mismo formato de estándar PNML [10] [11].

3.1.3 WoPeD

Es una aplicación de código libre de uso, de la Universidad de Karlsruhe (Alemania). Su objetivo principal es la modelación y simulación de flujos de trabajo, basado en redes de Petri. Tiene la capacidad de manejar estructuras de flujo de datos denominadas *workflows* y trabajarlas utilizando redes de Petri [12].

Esta, además de cumplir los estándares PNML [7], la estructura de *workflow* que se trabaja en esta aplicación también es reciclable para ser usada con otros tipos de lenguajes de modelado de procesos como *Yet Another Workflow Language* (YAWL) obteniendo así compatibilidad con una gran cantidad de herramientas.

3.1.4 Complementos para entornos de programación

Intentando concentrarnos en los productos semejantes a los que se quiere hacer, se buscaron complementos adaptados a entornos de programación. En este punto hubo muchos problemas en la búsqueda ya que, debido al estado obsoleto de la base de datos de programas de redes de Petri, solamente se encontró uno muy concreto denominado ITS Tools.

ITS Tools es un complemento de eclipse que trataba el diseño con redes de Petri, que a priori se debían de escribir mediante un lenguaje de programación creado por ellos específicamente para el uso de esta herramienta [13] .

El principal problema y con el que nos situamos en una posición nueva dentro del arte de las aplicaciones de redes de Petri es que no había complementos para Eclipse que tuvieran la capacidad de modelar visualmente líneas de producto de redes de Petri de forma flexible y admitiendo diversas configuraciones en función de un modelo característico.

3.2 Herramientas utilizadas para el desarrollo

Una vez entendida cual es la situación actual respecto a las aplicaciones basadas en la edición y el modelado de redes de Petri, es necesario poner en situación las herramientas que se utilizarán y el porqué de su uso.

El entorno de programación que se utilizará será Eclipse [14] en su versión de junio de 2019, la cual tiene las siguientes características instaladas:

- Eclipse Modeling Framework (EMF): es una herramienta de modelado y generación automática de código basada en las estructuras de datos generadas [15]. En nuestro caso lo utilizaremos para tratar unos metamodelos dados con la extensión *ecore*.
- OCL Classic: un editor de texto, que te permitía trabajar los metamodelos mediante *Object Constraint Language* (OCL) [16].
- Plug-in Development Environment (PDE): otra herramienta desarrollada por Eclipse Foundation, que sirve para el desarrollo de características nuevas para el entorno de programación Eclipse [17].
- Sirius: es un proyecto creado por Obeo para diseñar herramientas de modelado en Eclipse aprovechando las facilidades que trae EMF para ello [18].

4 Diseño

Una vez encontrado dónde se ubica nuestra aplicación dentro del esquema general marcado por el estado del arte, y explicadas las tecnologías que se utilizarán a lo largo del proyecto, trataremos de concretar la estructura de esta extensión para el IDE Eclipse. Para ello, se divide el proyecto en dos partes:

- Una herramienta de diseño de PNPL con los elementos de la misma asociada a diversas expresiones lógicas.
- Una extensión de Eclipse con la capacidad de tratar y procesar datos de las redes de Petri obtenidas mediante la herramienta de diseño.

Ambas partes deberán estar contenidas en un *Project Feature*, el cual tiene como objetivo juntar una cantidad determinada de *plug-in* para Eclipse en un solo proyecto para ser empaquetados. A lo largo de este capítulo señalaremos las decisiones que se han tomado a nivel de diseño de los diversos módulos que encontraremos en el proyecto.

4.1 Herramienta de diseño basada en Sirius

El primero de los módulos que hay que generar sería una herramienta básica para modelar redes de Petri que incorporen un sistema de variabilidad. La idea de esta herramienta es asociar elementos de una red de Petri (arcos, lugares o transiciones) con expresiones lógicas, basadas en un árbol de características (*feature model*). Para ello, se dispone de dos metamodelos que estarán representados en el anexo B del documento:

- Metamodelo de redes de Petri. Este diagrama fue obtenido a raíz de un metamodelo obtenido de la página PNML.org, la cual marca un estándar a la hora de crear aplicaciones basadas en redes de Petri con el objetivo de obtener una base genérica a la hora de generar dichas redes en los diversos programas, para que exista compatibilidad en ellos.
- Metamodelo de variabilidad. Un diagrama basado en expresiones lógicas con el cual añadir el uso de *features* al metamodelo de redes de Petri y poder así, representar diversas PNPL.

Por motivos de limitación impuestos por el uso de metamodelos y Sirius, se decidió juntar ambos metamodelos en uno y buscar la forma de unir sus clases de forma coherente. Esto se haría sin alterar demasiado sus clases y manteniendo así una posibilidad de separarlos de forma sencilla.

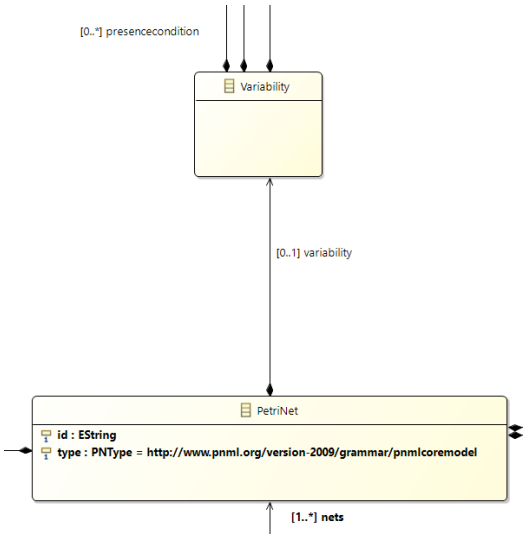

Figura 4-1. Unión entre ambos diagramas de clases.

Como se puede apreciar en la Figura 4-1, la forma más sencilla que se encontró para la unión de los dos diagramas de clases fue unir la clase raíz del diagrama de variabilidad de nombre `Variability` a la clase `PetriNet` en una relación de 0-1 para que, en el caso de que hubiera necesidad, poder utilizar el diseñador sin la parte de la variabilidad. Esta forma de unir ambos metamodelos, supone un cambio en el atributo `elements` dentro de la clase `PresenceCondition`. Inicialmente era un atributo de tipo referencia a una lista de elementos de tipo `EObject` con el que, en principio, valdría cualquier objeto de modelado basado en EMF [19]. En este caso, y basándonos en nuestro interés de unir los diagramas, se sustituyó por otra lista, pero con elementos de la clase `PNObject` procedente del metamodelo de clases de redes de Petri.

Además de estos cambios, se decidió añadir una serie de funciones de impresión para poder controlar los valores que se manejaban en la variabilidad. Así, para poder visualizar las expresiones lógicas derivadas del *feature model* que irían asociadas a los elementos de la PNPL que se está diseñando con la herramienta.

Respecto al modelado, nuestra implementación tiene seis elementos que están relacionados con la idea de crear PNPLs basadas en la aplicación de expresiones lógicas a *features*. Los elementos seleccionados para la generación serían los siguientes:

- Lugares: Correspondería dentro de nuestra unión de diagramas a la clase `Place`. Como se ha podido ver en el apartado de conocimientos previos, será representado con una circunferencia con el interior vacío.
- Transiciones: En el diagrama de clases correspondería a la clase `Transition`. La idea inicial es que se represente con un rectángulo, con la capacidad de rotar.
- Marcas o *tokens*: En este caso, el nombre de la clase a la que pertenecerá este tipo de elemento se llama `ToolInfo`. Esto se debe a que, dentro de los atributos correspondientes a la clase *Place* no se encontró ninguno que correspondiera correctamente a un número de *tokens* dentro de un lugar. Por ello, e intentando mantener un modelo lo más cercano al original, se escogió utilizar esta lista de elementos de tipo `ToolInfo`, que, en nuestra aplicación no iba a ser utilizada. Para representar cada *token*, se utilizará un punto negro en el interior de su lugar correspondiente pudiéndose meter todos los puntos que se deseen.

- Arcos: Para este elemento, dentro del diagrama encontramos una clase llamada `Arc` que pretende representar una relación entre nodos. Estos nodos serían un origen y un destino, y ambos serían referencias a instancias de la clase `Node`. De esta clase, heredan de forma indirecta tanto la clase `Place` como la clase `Transition`. La forma sencilla de representarlo sería con una flecha que fuera desde la representación del elemento origen, hasta la del elemento destino. Cabe destacar que es necesario en este punto que se cumpla aquella restricción ya mencionada en los conceptos básicos en el apartado 2.1 que explicaba que los arcos no se podían añadir entre dos elementos del mismo tipo.
- Expresiones: En este caso, para representar expresiones se decidió utilizar la clase `PresenceCondition`. Esta clase está compuesta por un atributo del tipo `Expression` que utilizaremos para representar una expresión. Se escogió esta clase debido a que `Expression` es una clase abstracta que, en la mayor parte de las clases que la heredan, está compuesta por uno o más elementos de su misma clase. Por ello, con `PresenceCondition` nos situamos en la raíz de un árbol que se formaría en el atributo ya mencionado. Utilizando las funciones de impresión mencionadas anteriormente, desde esta clase podremos obtener una expresión entera. Respecto a su representación, la idea es escribir de alguna manera la expresión y rodearla con alguna figura como un rectángulo para que quede clara su limitación.
- Relaciones entre expresiones y elementos dentro del diagrama: Es necesario saber a qué elemento o elementos de la PNPL que se está diseñando, se aplica cada una de las expresiones que se creen en el mismo diseño. Por ello aprovecharemos que la clase `PresenceCondition` también está compuesta por uno o más elementos que son referencias a instancias de tipo `PNObject`. Como `PresenceCondition` la utilizamos para la representación de expresiones, en este caso podríamos decir que la representación se haría mediante una línea que uniría expresiones con elementos y esta significaría un método `get` de la lista de elementos.

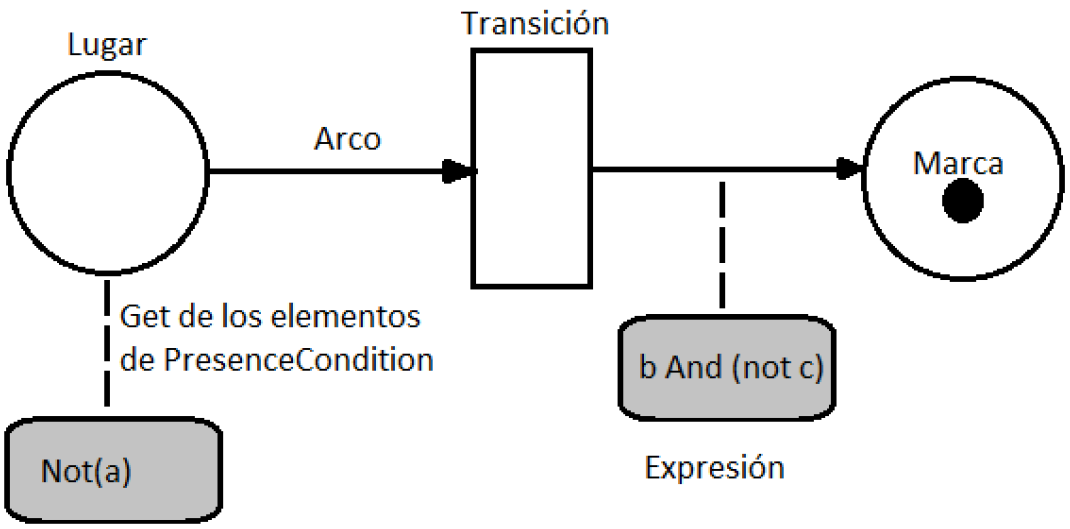

Figura 4-2. Idea básica de representación acorde a los elementos a diseñar.

La Figura 4-2 es un ejemplo sencillo del uso de todos los elementos que utilizaremos en nuestra herramienta y tratados de manera correcta. Es importante entender que toda expresión puede estar unida también con los tres elementos principales de la red de Petri tradicional.

Una vez realizado un diseño, este ha de ser guardado en un archivo con un formato interno de tipo XMI (basado en XML), intentando volver al diseño de redes de Petri original y obteniendo en otro fichero las expresiones aplicadas a cada elemento de la red que hayamos creado. En este sentido, se optó por separar el procesado de los archivos de la propia herramienta.

4.2 Complemento de procesado de archivos

Una vez guardados los datos y finalizado el trabajo que se vaya a realizar con la herramienta de diseño de PNPL basada en el generador de herramientas de Sirius, recordamos que esta característica de Eclipse es parte de un proyecto mayor que dando valores a las *features* trata de obtener varias redes de Petri de basadas en la inicial. Para ello se requiere una limpieza del fichero de salida que obtenemos mediante la herramienta.

El problema es que, una vez montada la red de Petri con expresiones lógicas en base a la unión de diagramas de clases, hay que separar los datos para obtener dos ficheros distintos ya que son necesarios para los algoritmos que analizarán las PNPLs :

- Un fichero con un formato de tipo XMI en el cual, solo estuviese la parte basada en el diagrama de redes de Petri.
- Un fichero con un formato con extensión vrb donde se guardarían el nombre de los ficheros correspondientes a la red de Petri y al *feature model* que se le aplicaría a dicha red y todas las expresiones aplicadas a la PNPL diseñada.

El primero de los casos debería ser muy sencillo puesto que, como se ha pretendido no cambiar en ningún momento el diagrama original de Pnmlcoremodel, a priori simplemente se debería poder buscar en el fichero final de la herramienta de diseño, la clase `Variability` que se añadió a la clase `PetriNet`, y eliminar todas las líneas que tuvieran que ver con ella (en lo que se refiere al formato XMI serían aquellas líneas comprendidas entre las etiquetas inicial y final de la clase `Variability`).

Por otro lado, en el segundo de los casos, requiere una extracción y volcado en otro fichero. Para ello se planteó la posibilidad de rediseñar el diagrama de clases relacionado con la variabilidad, haciendo cambios para que fuera más sencillo de trabajar desde el complemento para Eclipse.

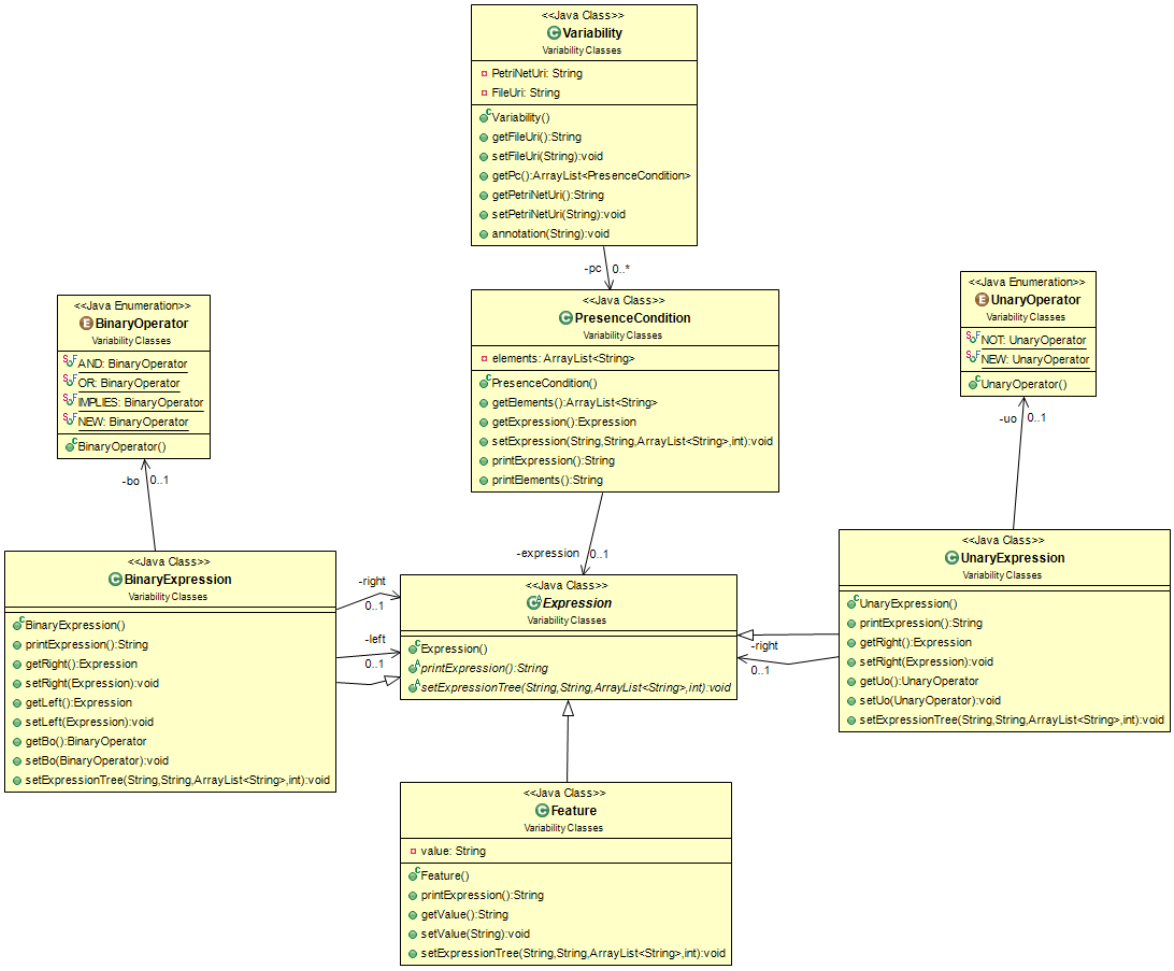

Figura 4-3. Diagrama de variabilidad con las clases simplificadas.

Como se puede apreciar en el diagrama de clases de la figura 4-3, respecto al diagrama de clases original ha desaparecido la clase `FileURI`, puesto que en este caso nos interesa tratarlo simplemente como cadenas de caracteres simples y en las enumeraciones de operadores ha aparecido el valor `NEW` simplemente para controlar errores. Además de estos cambios se añadieron funciones de inserción de datos en expresiones que se utilizarán para el volcado de datos del fichero original a las clases y también una serie de funciones para, con la clase `Variability` ya con todos los datos recolectados, volcarlos de manera correcta en el fichero de extensión vrb.

Finalmente, la clase donde se encuentra la función que se ejecutará cuando el complemento se ejecute, generará utilizando las funciones de la clase `Variability` el fichero de extensión vrb.

5 Desarrollo

En este apartado, se expondrá paso a paso y de forma detallada todo el trabajo que ha supuesto desarrollar la extensión de Eclipse. Para ello, las fases del desarrollo se han ordenado de forma cronológica.

5.1 Primeros pasos

Para poder desarrollar tanto la herramienta de diseño como el procesador de archivos, se tuvo que generar un proyecto de tipo *Ecore Modeling Project* el cual tendría un archivo de tipo ecore. En este archivo, que por motivos de facilidad su nombre es *pnmlcoremodel.ecore,* se volcaron los dos ficheros de extensión ecore correspondientes a los diagramas de clases iniciales ya mencionados en el apartado de diseño.

Mediante el editor OCL se estableció la unión de los elementos de ambos ecore, generando de forma sencilla la propiedad `variability` de tipo `Variability` dentro de la clase `PetriNet`. Dicha propiedad debía tener una restricción de composición para que se pudiese añadir más adelante a cualquier instancia de `PetriNet` y debía tener la posibilidad de existir o no, tal y como se muestra en la relación del diagrama de clases expuesta en la Figura 4-1.

Una vez desarrollada la relación, mediante el archivo de tipo genmodel que hay dentro del proyecto, se generó el todo el código relacionado con el proyecto, utilizando las opciones de *generate model code*, *generate edit code* y *generate editor code*. El código era necesario para poder trabajar con la herramienta puesto que, de forma interna Sirius haría llamadas a las clases y funciones generadas.

5.2 Desarrollo de la herramienta de diseño

Puesto que el desarrollo debe utilizar los paquetes de código mencionados anteriormente, se tuvo que ejecutar una segunda instancia de eclipse en tiempo de ejecución desde la primera. Para ello, se siguieron una serie de pasos indicados en el tutorial de Sirius [20] que básicamente consistían en ir dentro del menú de eclipse a *Run->Run configurations…* y en el interior escoger *Eclipse Application*. En este momento se generaría un nuevo elemento en las configuraciones de ejecución llamado *New_configuration.* Se escoge y se selecciona el botón *Run* para que se genere la nueva instancia de eclipse.

En esta nueva instancia de eclipse, puesto que se utilizan los proyectos abiertos dentro de la instancia inicial, se podrán generar archivos de tipo pnmlcoremodel, que nos servirán para generar ejemplos de uso del metamodelo utilizado previamente.

A partir de este momento, y durante todo el desarrollo del diseñador de PNPLs, se utilizarán distintas características del software denominado Sirius. En primer lugar, se crearon los dos tipos de proyectos que traía Sirius en sus componentes:

- *Modeling Project* (MP). Sirve para que a partir de un archivo basado en un modelo dado (en mi caso pnmlcoremodel), generar representaciones de éste mediante una herramienta hecha con Sirius.
- *Viewpoint Specification Project*(VSP). En este tipo de proyecto es en el que se crean herramientas de diseño asociadas un metamodelo.

Una vez creados, se añaden un archivo de tipo pnmlcoremodel. Para ello, hay que especificar la clase raíz y en nuestro caso fue `PetriNetDoc`. Esta clase fue escogida como raíz debido a que una restricción impuesta dentro del proyecto, ya que no dejaba crear esa clase como hijo de `PetriNet` y a la hora de validar el ejemplo, daba error. Este archivo fue introducido dentro del MP y en su interior se generó a mano un ejemplo sencillo añadiendo todos los tipos de elementos que se planeaba representar.

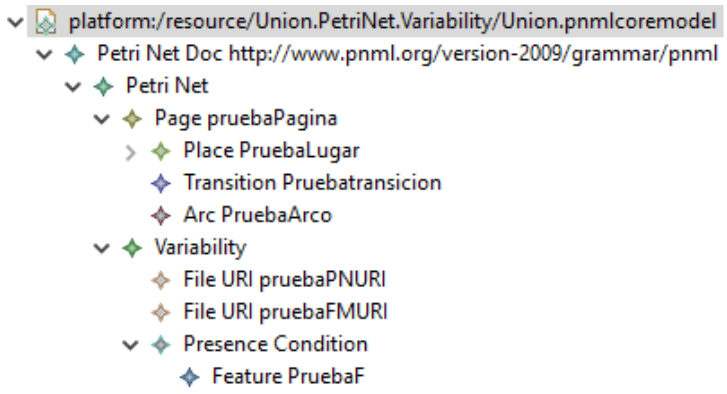

Figura 5-1. Ejemplo creado para el desarrollo de la herramienta.

Es importante destacar que, aunque no se muestre en la Figura 5-1, la *Presence Condition* generada para el ejemplo tiene como elemento al que se aplica, la instancia de la clase `Arc`. Por otro lado, también se han generado una serie de elementos que no se pretende representar, pero que sí que son necesarios a la hora de validar el ejemplo de cara a las restricciones que impone el modelo como las *File Uri*.

En este mismo instante, empecé a trabajar el VSP. Dentro de él, había un archivo concreto con un nombre puesto de forma automática por el creador de proyectos de Eclipse, de tipo *odesign*. En este archivo, es donde se crean y editan todos los elementos que debe tener la herramienta de diseño que queremos hacer. Para facilitar la creación se dividió el trabajo en dos partes: primero, la parte donde se representan los elementos y, segundo, la parte donde el usuario interacciona con la herramienta para poder crearlos, editarlos y eliminarlos de forma correcta.

5.2.1 Representación de los elementos del diagrama

Para la representación de los elementos que se pretende generar, en el archivo de extensión odesing, se crea una descripción de diagrama. En este momento se creará un subapartado que indica en sus propiedades el metamodelo y la clase raíz de la representación. En este caso, como metamodelo se elige pnmlcoremodel, y como nuestra clase raíz `PetriNetDoc`.

A partir de este momento, la idea es ir creando los elementos en función del tipo de representación que interese. Para ello, Sirius te da la posibilidad de crear los siguientes tipos de objetos utilizados en el diagrama:

- *Node*: Pretende representar la unidad más básica dentro del diagrama. Esto implica que tiene una representación directa en el diagrama, y no puede tener subnodos; pero, sin embargo, puede estar contenido en otros elementos. Respecto al estilo de su representación, es muy diverso. Desde una forma gráfica como es una elipse o un cuadrado, hasta incluso una imagen de extensión png.

- *Container*: Es un elemento cuya finalidad es delimitar un área en el cual habrá otros elementos de forma interna. Esos elementos podrán ser de cualquier tipo, aunque principalmente suelen ser más *containers* o *nodes* y servirán para representar hijos de la clase a la que representa el *container*. Estos elementos internos, podrán ser situados en su interior en forma de lista, pilas horizontales, pilas verticales o de forma libre sin ningún tipo de orden. El estilo, en este caso, solo podemos plantearlo de tres formas, la primera como un paralelogramo, la segunda como un gradiente y la tercera como una imagen. A priori Sirius nos brinda pocas posibilidades en el estilo de un *container*, pero al tener la posibilidad de insertar una imagen, el marco de posibilidades es infinito.
- *Element based edge*: Se usa para generar una relación entre elementos del diagrama. En este caso, dicha relación se corresponde a la representación de una instancia de una clase del metamodelo que es intermedia entre dos instancias que son nodos. Simplificando, un eje, al igual que los nodos, también representa una instancia de clase. En este caso su representación corresponde a una línea de unión entre una fuente y un destino y dicha línea tendrá muchas posibilidades de estilo como ser una flecha o una línea intermitente.
- *Relation based edge*: La idea es la misma que en *element based edge*, en este caso la relación no representa un elemento, sino representa un `get` de un *nodo/container* fuente, que devuelve otro elemento representado en el diagrama como destino. Su estilo es semejante al de *element based edge*.

Una vez explicados los elementos que pueden componer un diagrama, es el momento de aplicarlos a nuestro metamodelo. Durante el transcurso del Trabajo de Fin de Grado, se propusieron diversas posibilidades de representación hasta que se obtuvo la correcta. En esta memoria solo se expondrá esta última descartando así todos los errores de diseño cometidos.

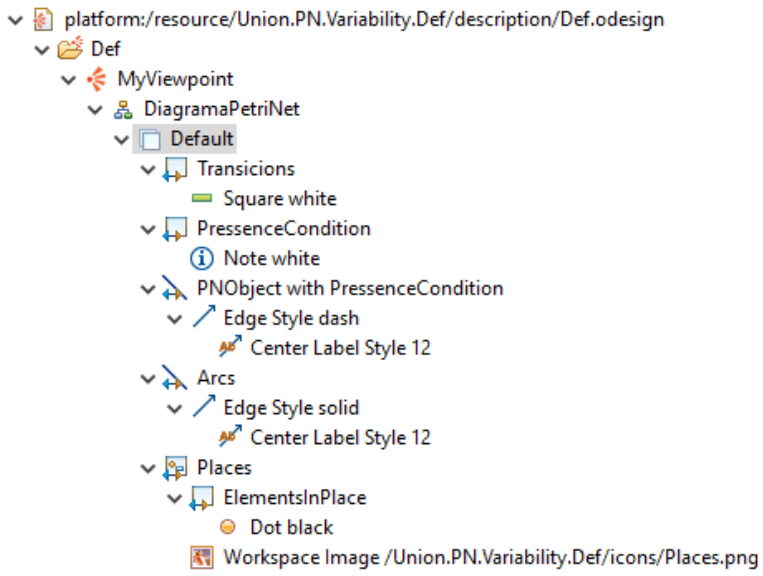

Figura 5-2. Árbol definitivo de la representación de los elementos del metamodelo.

5.2.1.1 Transiciones

Puesto que los elementos de la red de Petri original parecían más sencillos y las expresiones lógicas debían ser acopladas al sistema una vez que estos estuvieran creados en el mismo, se

decidió empezar por ellos. Concretamente se empezó por las transiciones porque era el único elemento totalmente independiente del resto.

Para su representación, como se puede ver en la Figura 5-2. Árbol definitivo de la representación de los elementos del metamodelo. mediante el símbolo a la izquierda de cada línea, se escogió *Node*. Esta decisión fue tomada ya que al ser independiente del resto de elementos y al ser una unidad básica dentro de la representación, *Node* era el tipo más correcto de los propuestos anteriormente.

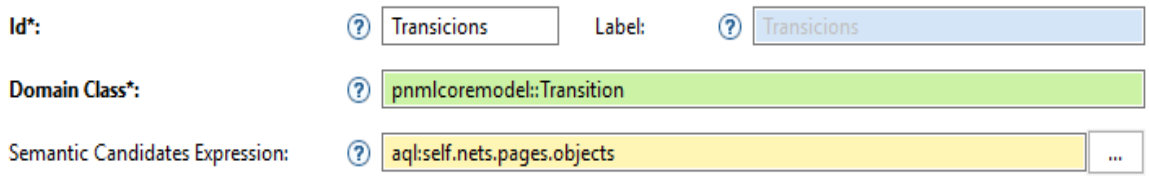

Figura 5-3. Propiedades del tipo *Node*, rellenadas acorde a las transiciones.

Dentro de las propiedades que propone *Node* para su uso, indicamos los siguientes parámetros:

- *Id*: Identificador para diferenciar de otros tipos de *Node*. En este caso escribí *Transitions*.
- *Domain Class*: Clase dentro del metamodelo que se va a representar. Rellené con *pnmlcoremodel::Transition*.
- *Semantic Cadidates Expression*: Este es un campo voluntario, en el cual Sirius mira a modo de consulta para no tener que mirar sobre todos los elementos y representarlos uno a uno. Se podría decir que es como una consulta y que, en un diagrama que fuese extremadamente grande, agilizaría tiempos de respuesta. En nuestro caso el lugar donde se sitúan las instancias de `Transition` está en una instancia de Page dentro de una de `PetriNet` situada como hija de la raíz. Como la búsqueda se inicia desde la raíz, la sentencia que se puso fue *aql:self.nets.pages.objects*. Esto haría que cualquier transición que no estuviese como elemento en la lista `objects` no sería representada por el diagrama.

Respecto al estilo, en diseño se planteó seguir con el modelo tradicional de transición, como se muestra en la Figura 4-2. No obstante, se encontraron problemas puesto que, los rectángulos como tal no estaban definidos y si fuera uno habría que tener en cuenta que debería poder rotarse. Como solución se planteó el cuadrado, que podía suplir todos estos problemas y, además también se ha visto usado en otras representaciones de redes de Petri. Además de la figura, las propiedades del estilo nos permitieron añadir una línea de texto con el id de la instancia de `Transition`.

5.2.1.2 Lugares

Cuando se planeó la representación de lugares, se hizo evidente que no podía ser semejante a la transición. Esto se debe a que los lugares pueden tener por dentro *tokens*. Por ello se utilizó un tipo de representación que cuadrase con la idea de "elementos dentro de elementos", y encontramos que Sirius nos proporciona el tipo *container*.

A continuación, volvemos a rellenar todas las propiedades que se nos piden, de la misma forma que se rellenó para las transiciones:

- *Id*: *Places*
- *Domain Class*: *pnmlcoremodel::Place*
- *Semantic Cadidates Expression*: Como los lugares se sitúan en la misma lista en la que se encuentran las transiciones, utilizamos la misma expresión: *aql:self.nets.pages.objects.*
- *Children Presentation*: es un campo adicional necesario para el tipo *container.* Sirve para saber cuál va a ser la disposición de los elementos internos a la hora de ser representados. En nuestro caso simplemente marcamos la casilla denominada *Free Form* puesto que simplemente estarán dentro, pero sin ningún orden real.

Acerca de la representación, en este caso el tipo *container* no ofrecía muchas posibilidades, y para generar un círculo dentro de nuestro diagrama se tuvo que generar una imagen de un círculo en una herramienta de diseño gráfico. Dicha imagen la añadiríamos al proyecto y se la daríamos mediante una ruta al estilo asociado al *container.*

5.2.1.3 Tokens

Para las *tokens* simplemente lo único que se tuvo en cuenta es que, su posición dentro del diagrama, debía ser en el interior de un lugar y, por tanto, no podían estar de forma libre. Partiendo de esta consideración se creó un nodo dentro del *container* de nombre *Places.* Las propiedades de nuestro nuevo nodo fueron las siguientes:

- *Id*: *ElementsInPlaces.* Este nombre fue escogido porque, a diferencia de los casos anteriores, no hay una relación entre el nombre de lo representado (*tokens*) y la clase.
- *Domain Class*: *pnmlcoremodel::ToolInfo*.
- *Semantic Cadidates Expression*: Para la expresión, valorando las diversas posibilidades que tenía para crearla, se pensó que *feature:eAllContents* era la opción más correcta. La razón de esto es que, utilizando *feature,* nos situábamos en nuestro contenedor de tipo *Place* concreto, y simplemente con *eAllContents* se buscaba mirar los elementos contenidos dentro de dicho contenedor.

Siguiendo la misma estructura que en los otros apartados, para el estilo se decidió utilizar puntos negros de un tamaño medio. Aquí surgió un problema relacionado con el tamaño de los lugares. El tamaño debía ser fijo, sin embargo, los *tokens* debían poder ser un valor ilimitado. Como no se encontró la forma de representar un solo elemento acompañado de un número, se decidió que excepcionalmente aquellos lugares con una gran cantidad de podrían ser más grandes o tener un *scroll* para recorrer su contenido.

5.2.1.4 Arcos

Para finalizar con la parte correspondiente al diseño de redes de Petri clásico, se genera una representación para los arcos, elementos que corresponderían a la unión entre un lugar y una transición, o viceversa. Para este caso, como existe una clase dentro del metamodelo llamada `Arcs` que representa la unión entre dos instancias de tipo `Node`, y tanto `Place` como `Transition` la heredan, en Sirius escogimos el tipo *Element based edge*.

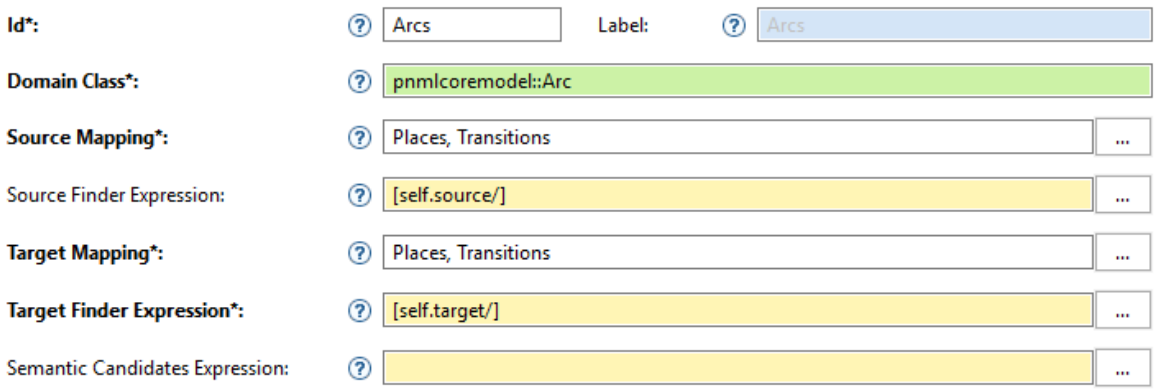

Figura 5-4. Propiedades del tipo *Element based edge*, rellenadas acorde al uso de arcos.

Tal y como se visualiza en la Figura 5-4, las propiedades difieren en algunos aspectos de las que pide un elemento de tipo *Node* cualquiera. Esto se debe principalmente a que este tipo, al intentar describir una unión, necesita saber su fuente y su destino. Considerando esto, los datos son los siguientes:

- *Id*: *Arcs.*
- *Domain Class*: *pnmlcoremodel::Arc.*
- *Source Mapping*: Define las componentes del diagrama que podían ser fuente. Como en una red de Petri los arcos van desde un lugar a una transición o desde una transición a un lugar, se rellenó el recuadro con *Places*, y *Transitions*.
- *Source Finder Expression*: De la misma forma que *Semantic Candidates Expression*, es una consulta para que la búsqueda de las instancias concretas sea más sencilla. Rellené este campo con *[self.source/]*, puesto que el propio elemento tenía un atributo que hacía referencia a un elemento origen.
- *Target Mapping*: Teniendo en cuenta lo explicado en *Source Mapping*, se rellenó con el mismo valor.
- *Target Finder Expression*: La expresión es *[self.target/]* ya que, al igual que pasaba con la fuente, hay un atributo que hace referencia al destino.

En el estilo, simplemente se marcó una flecha negra lo más semejante posible a la planteada en la Figura 4-2, a la que se le añadió un rotulo con el id de la instancia del arco representado.

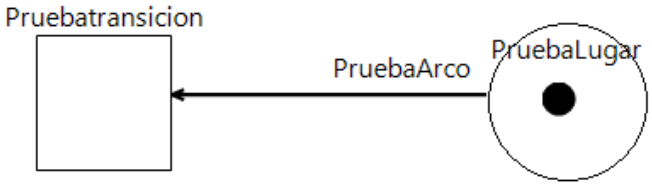

Figura 5-5. Representación actual del diseño.

En este punto, ya están creados todos los elementos de la red de Petri clásica. Por ello, se probó la representación actual del ejemplo planteado en la Figura 5-2. Para ello, en el proyecto MP accedimos al archivo llamado representations.aird y generamos una nueva representación utilizando el segundo proyecto. Como resultado se obtuvo la imagen de la Figura 5-5.

5.2.1.5 Expresiones

Se añadió de forma sencilla al diagrama mediante el uso del tipo *Node* ya que este es un elemento que inicialmente es independiente del resto. Como en otras ocasiones, rellenamos las propiedades:

- *Id*: *PresenceCondition.* Se escogió este nombre para que fuese el mismo que el nombre de la clase correspondiente. El objetivo es poder representar expresiones, pero, al depender de PresenceCondition, su representación no era directa.
- *Domain Class*: *pnmlcoremodel::PresenceCondition*
- *Semantic Cadidates Expression*: De la misma forma que se siguió la ruta para obtener los lugares y las transiciones, utilizamos una expresión para llegar al lugar donde se sitúan las instancias de PresenceCondition. Por ello, se introdujo la sentencia *aql:self.nets.variability.presencecondition.*

En este caso, el estilo tiene alguna particularidad. Se debía obtener una expresión completa desde la clase `PresenceCondition`. El problema surge cuando una expresión está formada por expresiones que pueden ser de diversos tipos y haber una gran cantidad de subexpresiones anidadas. Se podría decir que los usos que se le dan a la clase `Expression` en las clases que la heredan son con un sentido recursivo y formando una estructura cercana a un árbol. Se van sucediendo operaciones unarias o binarias hasta que en lo más profundo del árbol de instancias de `Expression`, donde se encuentran las instancias de la clase Feature, que actuarían como hojas (nodos terminales).

Se tomó como solución generar funciones que devolverían el árbol en forma de cadena de caracteres. Para ello, se volvió al código generado automáticamente del ecore en la primera instancia de Eclipse, y se añadió el método `printExpression` a diversas clases.

En la implementación de `PresenceCondition`, sería simplemente una llamada a la función correspondiente a la expresión raíz.

```
@Override
public String printExpression() {
	if (expression != null) {

		return expression.printExpression();
	}
	return "Rellenar en el arbol";
}
```

En `Expression` tendríamos un método abstracto, que sería implementado de formas distintas por las clases que lo heredan. El siguiente código presentado es la implementación en la clase `BinaryExpression.`

```
@Override
public String printExpression() {


	return "( " + left.printExpression()+ ' ' + operator.toString() +
		' ' + right.printExpression() + " )";
}
```

Como se puede observar, se van llamando continuamente a las funciones de las expresiones de la izquierda y la derecha, hasta que, finalmente se encuentra en lo más profundo una

instancia de la clase `Feature` la cual simplemente devuelve su atributo feature. Con ello, las devoluciones se van concatenando hasta llegar a `PresenceCondition` donde obtendremos cadena de la expresión.

Una vez implementados estos métodos en el metamodelo, se decidió para representar el estilo de *PresenceCondition* una nota. Se cambiaron las propiedades de la nota para que estuviese rodeada por un recuadro y así se pudieran ver sus límites, y me dispuse a ver la forma de llamar a la función de `printExpression`, ya que, las expresiones propuestas no te dejaban utilizarla.

En este momento se pensó que se podían usar las funciones de servicios de Sirius. Estas son métodos que crean los usuarios para su herramienta y que tienen a su disposición el metamodelo. Por ello, se creó una función sencilla llamada `expression` que simplemente llamaba a `printExpression` y retornaba su devolución.

Simplemente se añadió en las propiedades de la nota, más concretamente en *Label Expression* que será el valor que se vea en el diagrama en el interior de la nota, *service:expression()*.

5.2.1.6 Relacion Expresion-Elemento red de Petri

A diferencia de la relación que simboliza la clase `Arc`, en este caso no hay ningún tipo de elemento que represente la relación entre las expresiones y los objetos que conforman la red de Petri. Sin embargo, uno de los motivos por el que se escogió la clase `PresenceCondition` a la hora de representar expresiones, era porque tenía una lista con los elementos de la red de Petri a los que se asociaba su expresión. Por ello, en Sirius, escogimos el tipo *Relation based edge*, ya que en este caso simplemente generaríamos las relaciones en el diagrama mediante la obtención de la lista.

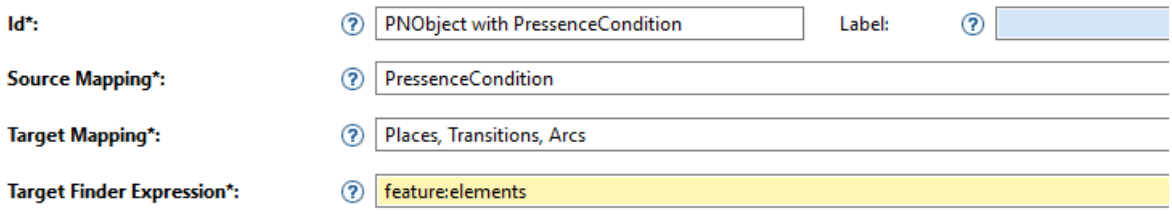

Figura 5-6. Propiedades del tipo *Relation based edge*, rellenadas acorde a la unión entre las expresiones y los elementos de la red de Petri clásica.

La forma de rellenar las propiedades de este tipo es semejante a la usada en *Element based edge*, la única diferencia es que no existe el campo *Source Finder Expression* debido a, que se genera a partir del elemento fuente. Haciendo un repaso al resto de propiedades:

- *Id*: *PNObject with PresenceCondition.*
- *Source Mapping*: *PresenceCondition.*
- *Target Mapping*: *Places, Transitions, Arcs.*
- *Target Finder Expression*: La expresión es *feature:elements* porque simplemente teníamos que obtener la lista de elementos.

En el estilo, se llegó a la conclusión de que, aunque hubiera fuente y destino diferenciados, para el diseño que habíamos hecho no cuadraba hacer una flecha ya que se quería representar de forma distinta a los arcos. Por ello, con un color más grisáceo, se hicieron líneas discontinuas sin dirección alguna.

Finalizada la creación de este tipo de relaciones, podemos decir que el diseño de los elementos en el diagrama ha sido finalizado. Por ello, volvemos a ver nuestro diagrama actualizado.

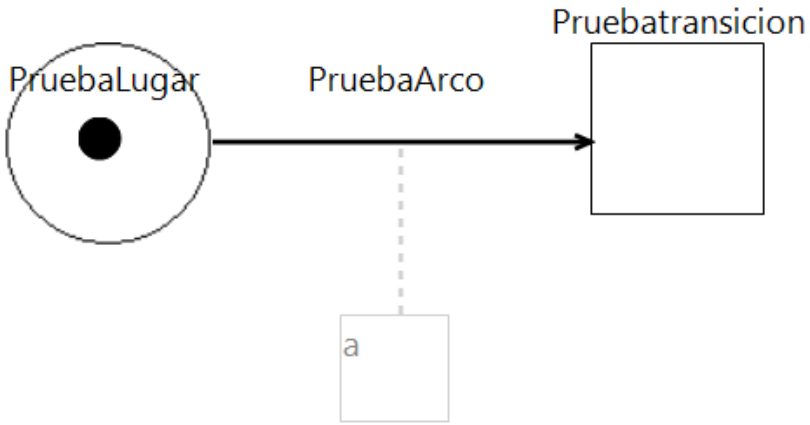

Figura 5-7. Representación final de los elementos.

Respecto a la Figura 5-5, en esta Figura 5-7 se movieron los elementos para ver un orden más lógico. Una vez generada la representación de todos los tipos de elementos, se debe generar una herramienta de creación para que sea más simple e intuitivo que ir rellenando el fichero que se representa.

5.2.2 Herramienta de creación de los elementos en el diagrama.

De la misma forma que con la representación de los elementos, dentro de nuestro archivo de tipo odesign se generó una nueva sección, donde tendremos botones que servirían para crear los elementos.

Para la programación de estas herramientas, hay definida una serie de operaciones cercanas a cualquier lenguaje imperativo como Java. Se puede ver cuáles son y cómo se usan en la documentación de Sirius [21]

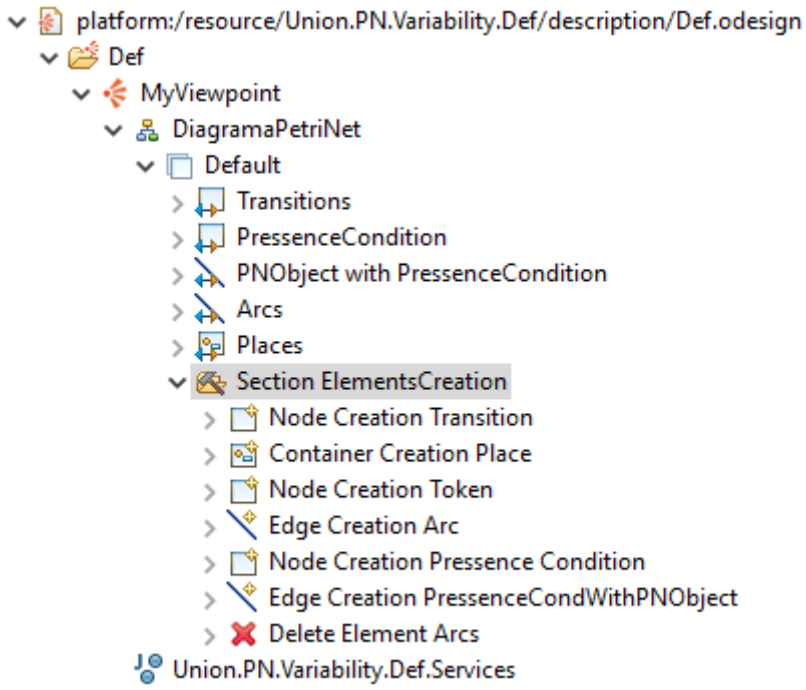

Figura 5-8. Árbol en el que se incluyen las opciones de modelado de la herramienta.

Para explicar el procedimiento que se ha seguido a la hora de generar las herramientas de la Figura 5-8, éstas se agruparán en dos tipos en base a si hay necesidad de que haya elementos creados anteriormente o no.

5.2.2.1 Elementos independientes.

Los elementos independientes para nuestro diseño son las instancias de las clases `Place`, `Transition` y `PresenceCondition` ya que ninguno de estos elementos depende de los demás dentro del diagrama. La principal razón por la que se han separado del resto de los elementos es porque, al ser independientes, se deberían poder crear en cualquier momento una vez iniciada la representación. Por ello, cuando no hay ni un solo elemento creado en el fichero, se deben generar todos los elementos intermedios, puesto que las instancias representadas en el modelo deben tener como ancestro a la instancia raíz. Para hacer este control de errores, dentro de la creación de todos los elementos independientes, se comprobó que existía una instancia de `PetriNet`. Si no era el caso, se debía crear dicha instancia, y sus instancias hijas:

- Una instancia de `Page`. Cuando llegamos a este punto, podemos crear los elementos de la red de Petri original.
- Una instancia de `Variability`. Con esto, podemos crear la parte de las expresiones lógicas sin problemas. En este caso, también habría que crear las dos instancias de `FileUri` correspondientes puesto que hay una restricción OCL que nos impide validar el modelo sin ellas.

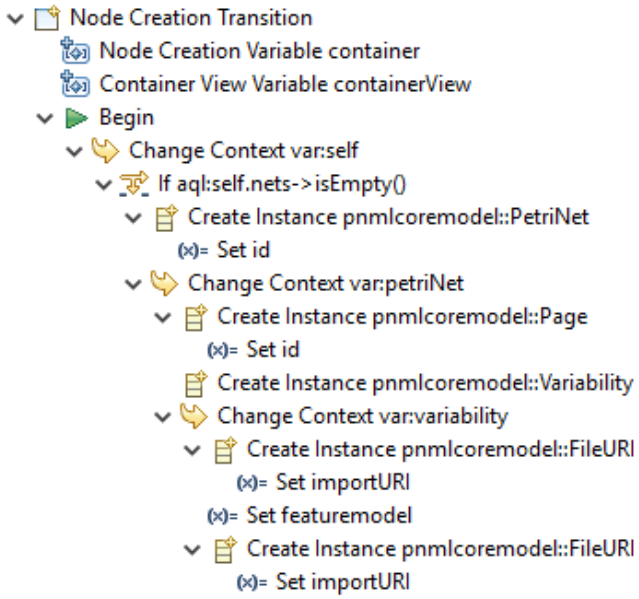

Figura 5-9. Parte común en la creación de elementos independientes.

Una vez explicada esta parte común, hay una parte de código que es específica para cada caso.

Lugares

Para los lugares, había que acceder desde la raíz hasta el lugar donde se situaban las instancias de la clase `Place`. Por ello, la forma, más sencilla que se encontró, fue acceder mediante bucles anidados. Como en la lista `petrinets` dentro de `PetriNetDoc` solo hay una instancia, solamente ejecuta acciones sobre ese elemento. Lo mismo pasa cuando miramos la lista `pages`.

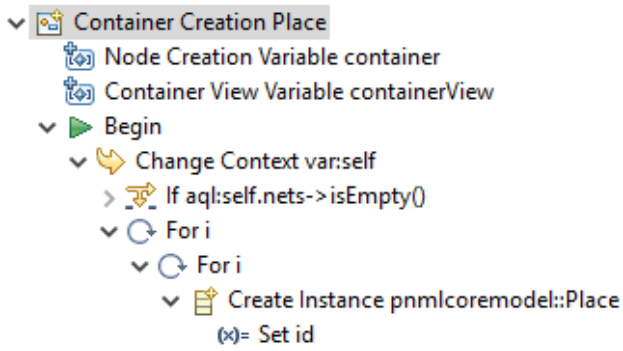

Figura 5-10. Árbol que representa la creación de una instancia de *Place*.

Para este caso concreto, se crea la instancia y se le añade un id. Para crear este id, volvimos a usar los servicios para crear métodos de Sirius. En el archivo Services.java, se creó una variable privada y estática llamada `idPlaces`, que irá aumentando de tamaño conforme se vayan generando instancias de `Place`.

```
public int idPlace(Place p) {
      idPlaces++;
      return idPlaces;
```

```
}
```

La importancia ir variando el id que se le daba a cada instancia, residía en la existencia de una restricción que obligaba a tener distintos ids en los elementos que se iban creando e insertando en la lista `objects` de una página.

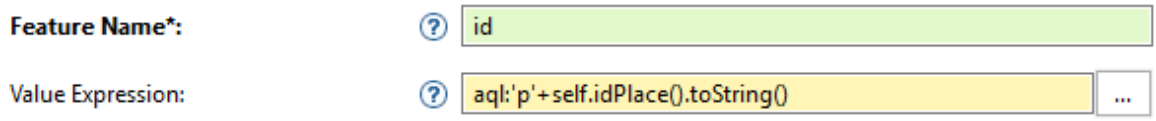

Figura 5-11. Asignación de la función al id de la instancia de *Place*.

En la Figura 5-11, se puede ver como se añade al inicio del valor una 'p' haciendo referencia a la *place* y luego se añade el número, con la idea de que las instancias de distinto tipo no tengan el mismo id.

Transiciones.

Para las transiciones se tiene que acceder desde la raíz hasta el lugar donde se sitúan las instancias de la clase `Transition`. Como se sitúan en la misma lista de elementos que las instancias de Place, hacen el mismo recorrido mediante bucles anidados.

La creación del id es semejante. Se utiliza una variable llamada `idTransitions` cuyo código es semejante al propuesto para los lugares. En la creación de la instancia, cuando damos valor al id, lo acompañamos en este caso de una 't'.

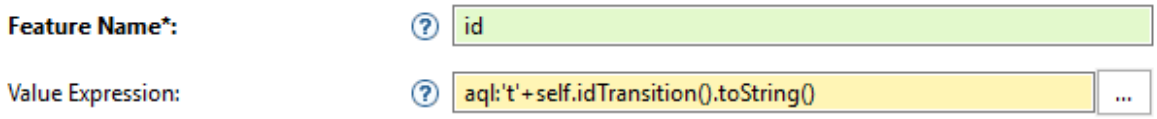

Figura 5-12. Asignación de la función al id de la instancia de Transition.

Expresiones

En el caso de las expresiones, desde la raíz del modelo, tendremos que ir a la instancia de la clase `Variability` cada vez que se use la herramienta de creación. Para ello, volvemos a entrar en la única instancia de `PetriNet`, y desde esta, cambiamos de contexto al interior del atributo `variability`.

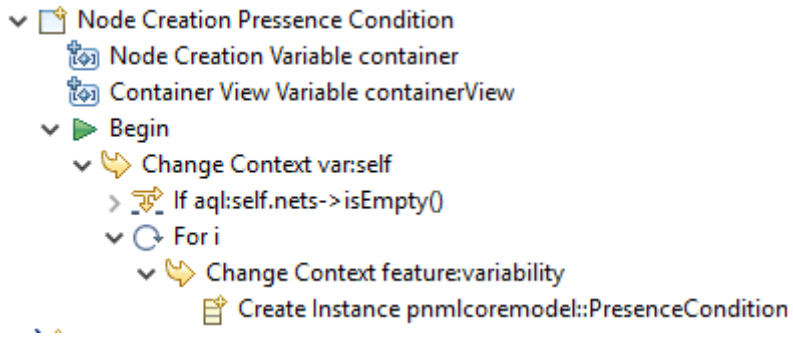

Figura 5-13. Árbol que representa la creación de una instancia de *PresenceCondition*.

Para la creación de la instancia de `PresenceCondition`, no fue necesario asignar ningún valor a sus atributos. Esto se debe a que, para el atributo `elements` se irán adhiriendo elementos mediante la creación de las relaciones correspondientes, y para `expression`, se debía crear a través del fichero pnmlcoremodel debido a su complejidad. Por ello, cuando se añadieron las funciones `printExpression` al metamodelo, se controló el caso de que estuviera a *null* devolviendo la cadena "Rellenar el árbol". Esta sentencia aparecería en el diagrama cuando se crease una `PresenceCondition`.

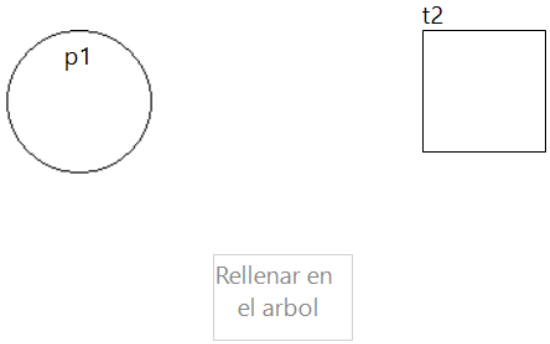

Figura 5-14. Elementos independientes creados con la herramienta.

El significado que tiene la frase "Rellenar el árbol", es que las expresiones no se pueden crear mediante la herramienta. Por ello, si uno quiere obtener una expresión dibujada dentro del diagrama, deberá ir al archivo de extensión pnmlcoremodel, y generar todos los hijos necesarios para hacer una expresión completa. Es importante recordar que las instancias de la clase `Feature` son los nodos terminales.

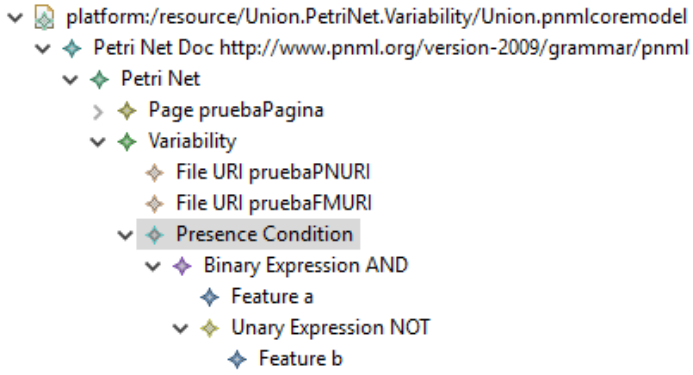

Figura 5-15. Árbol con una expresión creada.

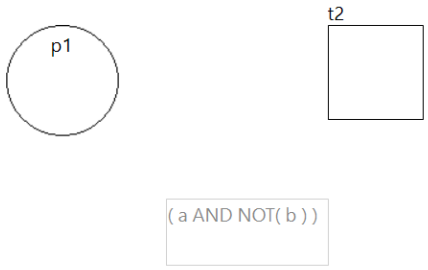

Figura 5-16. Adición de la expresión en el diagrama.

Una vez se ha creado y validado la expresión, podemos volver al diagrama. Ahora, al ser el atributo `expression` no nulo, se debería apreciar la expresión lógica como en la Figura 5-16.

5.2.2.2 Elementos dependientes

Los elementos dependientes, son aquellos que, para poder ser creados y representados, tienen que haberse cumplido algunas condiciones que tienen que ver con los elementos independientes. Adicionalmente, aparte de la creación de elementos, también se generó la eliminación de algunos de ellos.

Tokens

Para que se pudiesen crear las instancias de `ToolInfo`, era necesario que se hubiera creado ya al menos un lugar. Como su diseño se generó dentro de su contenedor, Sirius ya detectaba de por sí que no podía crear las instancias en cualquier lugar (sale un símbolo de prohibido cuando se intenta crear fuera de un lugar).

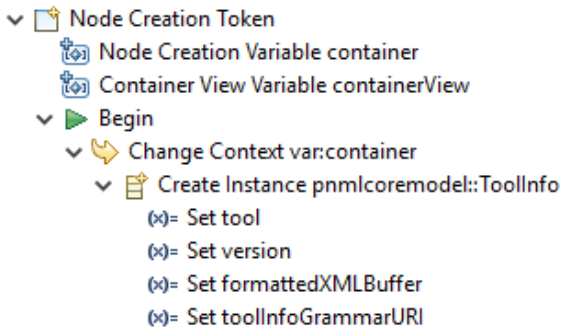

Figura 5-17. Árbol de la creación de un *token*.

Lo más interesante que se ve en la Figura 5-17 es el cambio de contexto a *var:container*. Con esta expresión, las siguientes operaciones anidadas se sitúan dentro del *container*, que es el lugar escogido para crear el *token*. Una vez en el contexto, se crea la instancia y se da valor a todos sus campos manteniendo así una validación correcta del modelo.

Arcos

Para la creación de arcos, debían estar creadas al menos una transición y un lugar. En este caso es importante recordar que, aunque estemos tratando de generar un eje, éste representa a una instancia de una clase que tiene en sus atributos los elementos que une.

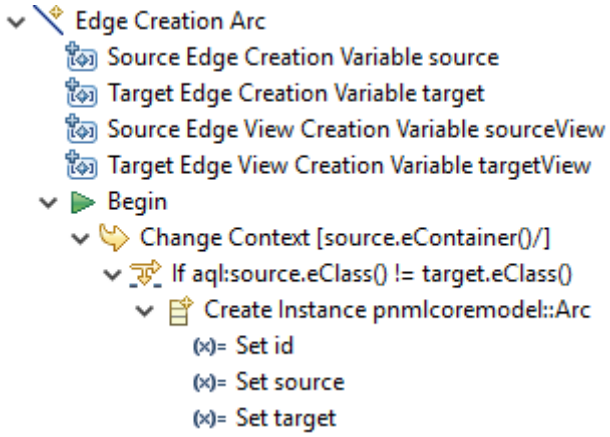

Figura 5-18. Árbol de la creación de un arco.

En la creación de un arco, lo más importante a destacar, es que no se podían generar entre elementos del mismo tipo de clase debido las restricciones que tienen las redes de Petri. Por ello, se generó una sentencia condicional que solo te permitía crear un arco si se daba el caso de que las clases de las instancias no eran iguales.

Una vez llegada la creación del id, su generación sería exactamente igual a la de las transiciones y los lugares. Existe una variable estática privada llamada `idArcs` que se va actualizando cuando se llama al método `idArc`.

Unión entre expresiones y elementos de la red de Petri

La creación de esta unión era la más sencilla de todas. Puesto que se representaba en el diagrama cada eje como un elemento de una lista, y los elementos a insertar en dicha lista tenían que estar ya creados, simplemente era hacer una llamada al método set junto al elemento a añadir.

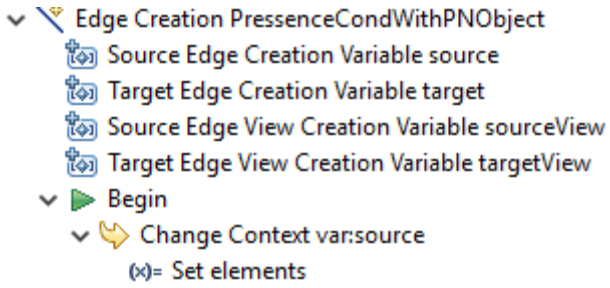

Figura 5-19. Árbol de la creación de la unión entre instancias de PresenceCondition y de elementos de la red de Petri.

Como en este caso la variable fuente debe ser siempre la instancia de `PresenceCondition`, se pudo cambiar a su contexto y hacer un set sobre la lista `elements` con la variable llamada *target*.

Eliminación de arcos en la eliminación de transiciones y lugares.

Adicionalmente, a parte de las herramientas de creación, se comprobó que las instancias de `Arcs` cuando se eliminaba una de `Place` o de `Transition`, se perdía su representación, pero seguían existiendo. Por dicho motivo, se sobrescribió la eliminación genérica de estos dos elementos.

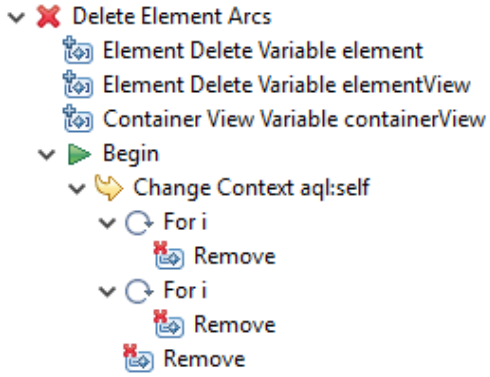

Figura 5-20. Árbol de la eliminación de arcos cuando se eliminan lugares o transiciones.

Para la eliminación de las instancias de `Arc`, como las clase `Place` y `Transition` heredan de `Node`, tienen unas listas que hacen referencia a arcos entrantes y salientes. Conociendo esto, el procedimiento que hay que seguir es recorrer ambas listas eliminando todos sus elementos. Finalmente, cuando se eliminan todos los arcos entrantes y salientes, hay que eliminar el elemento en sí. [6]

En este momento podemos decir que la herramienta está acabada ya que es posible crear y utilizar todos los elementos que compondrán el diagrama y además la generación cumplirá las restricciones impuestas por el metamodelo.

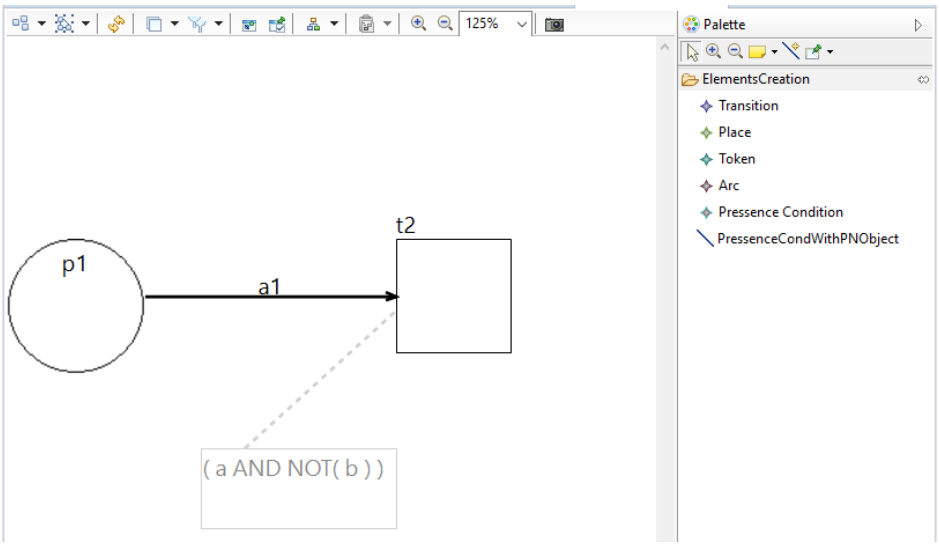

Figura 5-21. Diseñador con un ejemplo concreto hecho mediante las propias herramientas.

5.3 Complemento de procesado de archivos.

Como ya se sabe, esta herramienta de modelado, sirve para crear PNPLs, en las cuales no hay una ejecución y tampoco se diferencian unas líneas de producto de otras. Sin embargo, se usará para hacer un análisis profundo de las redes que se pueden generar. Estos algoritmos son externos a la aplicación y necesitan una serie de archivos de entrada [4] [5]. Para ello, necesitamos a raíz del archivo de extensión pnmlcoremodel, separar la variabilidad y la red de Petri tradicional en archivos distintos:

- El archivo relacionado con redes de Petri, seguirá un formato XMI semejante al creado desde la herramienta de Sirius, pero habrá que eliminar todas las líneas relacionadas con la variabilidad.
- El archivo que contendrá la variabilidad y sus relaciones con los elementos de la red de Petri. Este archivo tendrá una extensión de tipo vrb y simplemente tendrá impresiones de las instancias de `PresenceCondition`.

Para la creación de este complemento se generó en la segunda instancia de eclipse (aquella donde se hizo la herramienta de diseño con Sirius) un proyecto de tipo *Plug-in Project*, donde tal y como se enseñó en la asignatura de Desarrollo Automatizado de Software, se añadió una pestaña de menú al proyecto.

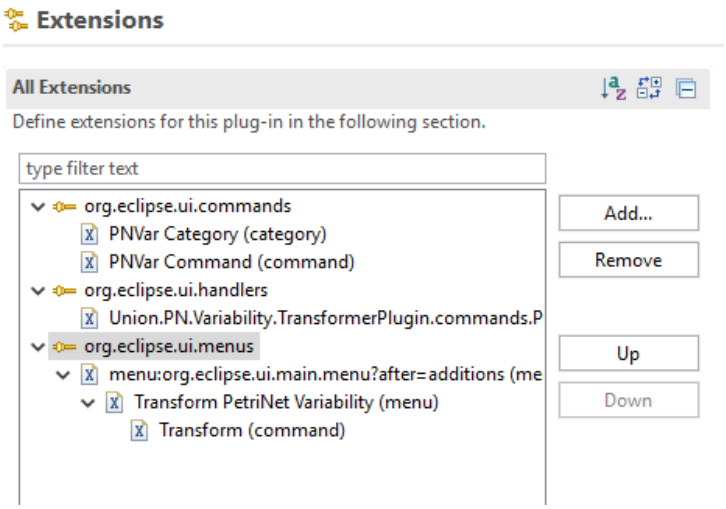

Figura 5-22. Adición de los archivos que generarían la herramienta del menú.

Añadiendo estas extensiones, se generaría una clase `PNVarHandler` que extiende a `AbstractHandler`, una clase que tiene un método abstracto llamado `execute`. Este método, es aquel que se ejecutará cuando en el menú de Eclipse, se siga la ruta *Transform PetriNet Variability->Transform*. A continuación, se explicará de forma sencilla el proceso de formación de este método.

5.3.1 Menú de selección del fichero que se quiere transformar.

La forma en la que se ha añadido este complemento haría que esta opción de usarlo estaría presente independientemente de la perspectiva o del proyecto en el que se estuviese trabajando. Por ello, se optó por generar un menú de selección que nos diese la posibilidad de escoger el archivo concreto que del que se quieren obtener los dos nuevos ficheros.
Para la obtención de los archivos del *workspace* se utilizó el siguiente código:

```
File root = ResourcesPlugin.getWorkspace().getRoot().getLocation().toFile();

ArrayList<File> listaArchivos = new ArrayList<File>();

try {
	Files.walk(Paths.get(root.getAbsolutePath()))
	.filter(x -> x.toString().endsWith(".pnmlcoremodel"))
	.forEach(x -> listaArchivos.add(x.toFile()));
```

```
} catch (IOException e1) {
		e1.printStackTrace();
}
```

Cuando generamos la variable `root` obtenemos la ruta donde se sitúa el *workspace*. Esta ruta la insertamos dentro de la expresión lambda generada dentro del `try-catch`, y básicamente se hace una búsqueda recursiva filtrando y guardando los archivos que acaben en la extensión de pnmlcoremodel.

Una vez obtenidos, estos archivos se irán volcando en una variable de tipo `String[]` necesaria para generar el menú de selección.

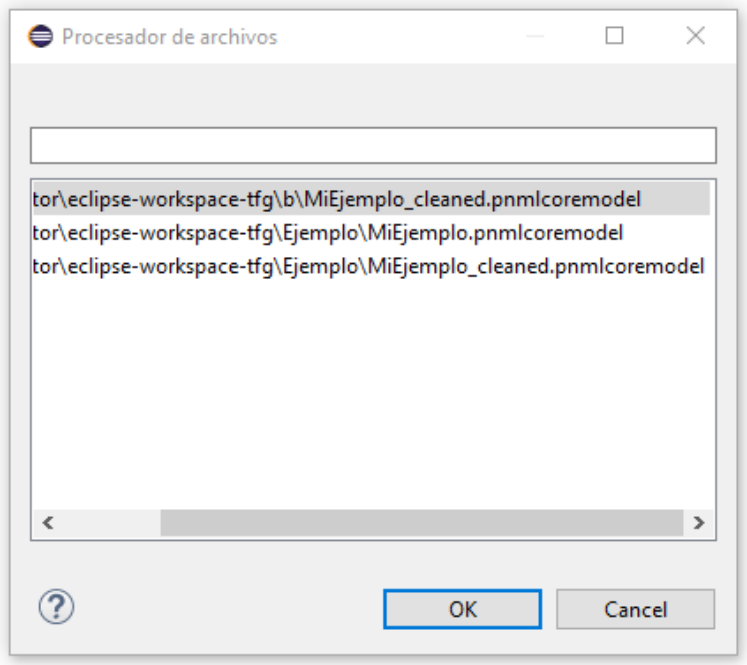

Figura 5-23. Ejemplo del menú de selección del archivo que se va a procesar.

Desde el menú mostrado en la Figura 5-23, se seleccionaba un archivo, que el método `execute` estaba esperando. Una vez seleccionado, se tendría que hacer su procesado y la generación de los dos ficheros.

5.3.2 Procesado del fichero base y generación de los nuevos ficheros.

Una vez tenemos el fichero que queremos tratar, se hace un recorrido básico por el mismo, leyendo línea a línea. Debemos tener en cuenta que, con su contenido hay que generar dos ficheros.

5.3.2.1 Fichero de pnmlcoremodel original

Para generar este archivo, debemos simplemente eliminar la parte de variabilidad del fichero con el modelo original creado con la herramienta de desarrollo. Por este motivo, se creó un archivo del mismo nombre acompañado al final por la cadena "_cleaned". Dentro del bucle de lectura del archivo, se utilizaría una *flag* para saber si dentro del fichero nos situamos en la parte de la variabilidad o no.

```
int flagLimpieza = 0;
while (scanner.hasNextLine()) {
	String aux = scanner.nextLine();
	if (aux.contains("<variability>")) {
		flagLimpieza = 1;
		…		//codigo intermedio donde se trata la variabilidad
	}
	if (flagLimpieza == 0) {
		writer.write(aux);
		writer.write("\n");
	}

	if (aux.contains("</variability>")) {
		flagLimpieza = 0;
	}
}
```

En el código anterior se pueden ver una serie de condiciones donde se comprueba la situación de la lectura línea a línea del fichero original. Si la lectura está fuera de la parte de variabilidad, vamos copiándolo en nuestro fichero limpio.

5.3.2.2 Fichero que contiene la variabilidad y su relación con la red de Petri

La formación de este archivo tenía una peculiaridad. Revisando el fichero original de Sirius, se deduce que se deben generar unas clases simplificadas basadas en el diagrama de variabilidad del anexo B, ya que el código generado automáticamente del metamodelo no aportaba la funcionalidad necesaria.

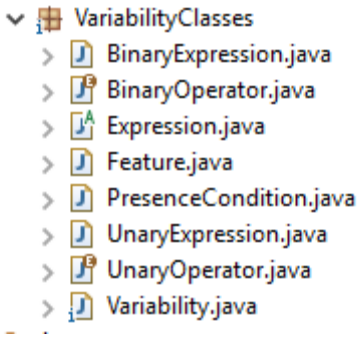

Figura 5-24. Paquete que contiene las clases reducidas de la variabilidad.

Respecto al trabajo que se hizo desde la lectura del fichero obtenido en Sirius, en el momento en el que se encontraba la etiqueta de variabilidad, se generaba otro bucle, que seguía leyendo línea a línea siempre que siguiésemos en este campo. La inserción de la mayor parte de los elementos es sencilla, si en la línea correspondiente se encuentra las etiquetas correspondientes a cada clase, se genera y se crea. Esto sucederá en todos los casos menos en el de la creación de la expresión.

```
//primer caso que inicia en expression, buscamos etiquetas y generamos con
//setExpression

if (aux2.contains("<expression")) {
```

```
	if (aux2.contains("Binary")) {
		if (aux2.contains("operator=")) {
			String value = aux2.split("operator=")[1].split("\"")[1];

			var.getPc().get(flagListPresence)
			.setExpression("Binary", value, null, -1);

		} else {
			var.getPc().get(flagListPresence)
			.setExpression("Binary", "AND", null, -1);
		}

	} else if (aux2.contains("Unary")) {

	…

	} else if (aux2.contains(":Feature")) {

	…

	}
```

El código anterior forma parte de la creación de la raíz del árbol de expresiones que, con la unión de todas se forma una expresión completa. Solo se ha añadido la parte de la expresión binaria en el caso de que el elemento a insertar sea la raíz del árbol de expresiones, pero en los tres casos se trabaja llamando a la función `setExpression`, que era una compleja función que recibe como argumento un tipo de expresión, la operación o valor de la expresión, una lista con el camino a seguir dentro del árbol para insertar la nueva expresión en su sitio correspondiente, y la profundidad de la posición que estamos buscando.

```
if(expression == null) {
	if(type.equals("Binary")) {

	BinaryExpression b = new BinaryExpression();

	if(valor.equals("AND"))
		b.setBo(BinaryOperator.AND);
	if(valor.equals("OR"))
		b.setBo(BinaryOperator.OR);
	if(valor.equals("IMPLIES"))
		b.setBo(BinaryOperator.IMPLIES);

	expression = b;
	return;
	}

	…	//El resto de casos (expresiones unarias y features)
}
		//si esta relleno por dentro la expression
else {
	expression.setExpressionTree(type, valor, ruta, recorrido);
	return;
}
```

El código anterior, refleja la función desde la clase `PresenceCondition`. Simplemente detecta si el atributo expresión es nulo, y si es así, verifica el valor de los argumentos para

crear la que expresión corresponde. En el caso contrario, le pasa los argumentos a la funcion `setExpressionTree`, la cual comprobará los componentes a la izquierda o a la derecha, en función del tipo de elemento y del camino que se busque, y seguirá profundizando el árbol hasta que se localice un atributo de tipo `Expression` que sea nulo, donde insertará la nueva instancia que se cree en función de los argumentos.

Una vez acabada toda la creación de elementos dentro de las etiquetas de variabilidad se generaba el fichero mediante una función de impresión en la clase `Variability` llamada `annotation`.

```
public void annotation(String nombreArchivo) {

	try {
		FileWriter myWriter = new FileWriter(nombreArchivo);

		myWriter.write("pn " + PetriNetUri + "\n");
		myWriter.write("fm " + FileUri + "\n");

		myWriter.write("\n");

		for (PresenceCondition p : pc) {
			myWriter.write("PC for " + p.printElements() +
				"= " + p.printExpression() + "\n");
		}
		myWriter.close();
	} catch (IOException e) {

		e.printStackTrace();
	}
}
```

Con este código incluyendo las funciones `printElements` y `printExpression` que consistían simplemente cadenas acordes a un formato dado para que el fichero generado, se pudiera utilizar en otra aplicación, se finaliza el desarrollo del complemento.

En este momento y siguiendo uno de los tutoriales de Sirius [20], se creó un proyecto de tipo *Feature Project* para incluir en él todos los proyectos necesarios para que todo funcionase correctamente.

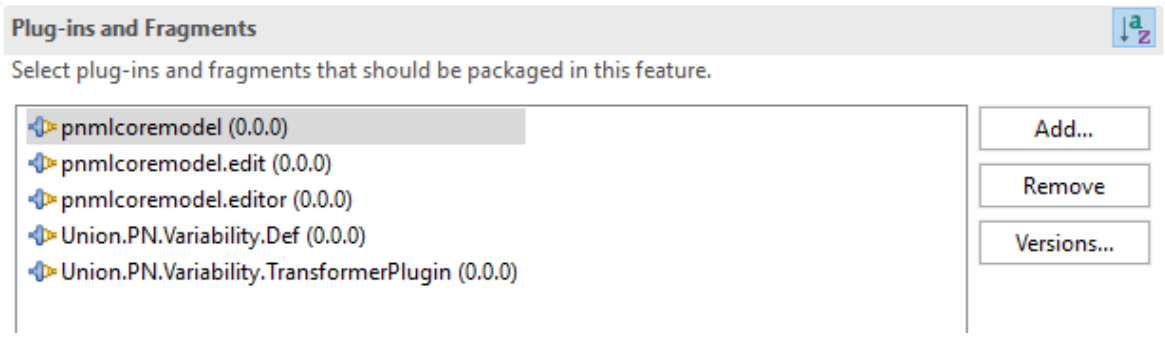

Figura 5-25. Elementos añadidos al *Project Feature* generado.

Finalmente, se creó un nuevo proyecto de tipo *Update Site Project*, cuyo objetivo era unificar todas las características del proyecto anterior y convertirlo en un comprimido de tipo zip instalable en Eclipse.

6 Integración, pruebas y resultados

En este Capítulo se muestran las pruebas realizadas para demostrar el correcto funcionamiento de la herramienta desarrollada. También se probó el complemento final en una instalación "limpia" de Eclipse 2019-06, y se pudo comprobar su correcto funcionamiento. La mayor parte de las pruebas se pueden dividir de la misma forma en la que se dividió la fase de desarrollo.

6.1 Desarrollo de la herramienta de diseño

A lo largo de toda la creación del diseñador de redes de Petri, se fueron probando todos los elementos cada vez que uno era terminado para comprobar que su estilo se correspondía con la idea inicial expuesta en el desarrollo y que su funcionabilidad fuese semejante con la impuesta en los metamodelos dados y con los conceptos básicos de redes de Petri.

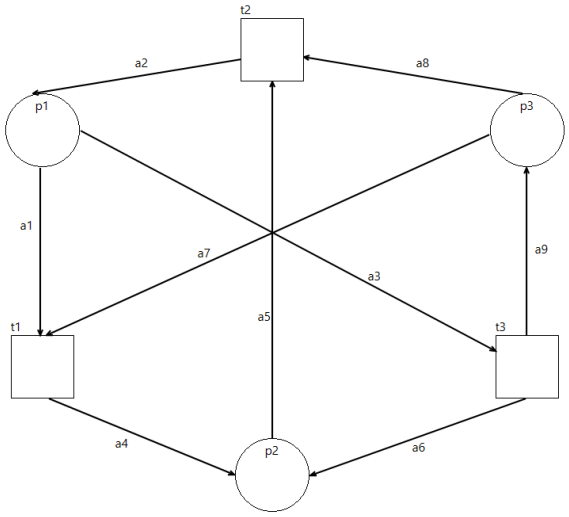

Figura 6-1. Prueba sencilla de red de Petri, una vez se diseñaron las herramientas de creación.

Además de estas pruebas a lo largo del proceso de creación, se hizo una prueba exhaustiva final, que consistía en generar una red de Petri basada en una publicación [4]. Se pudo comprobar que el esquema era validado correctamente y que no se encontraba ningún tipo de error. Con esto, se demostraba que la herramienta de diseño era completamente funcional y que atendía a las necesidades que se planteaban.

No obstante, al realizar pruebas que tenían que ver con eclipse, se descubrió que cuando se cerraba eclipse y se volvía a abrir, al generar elementos en un diagrama ya existente, las variables de ids del diseñador volvían a su valor inicial y esto generaba problemas en la validación ya que todos los lugares, transiciones y arcos debían tener un id distinto. El id se podía cambiar a mano desde las propiedades de cada elemento dentro del diseñador, así que no se le dio mucha importancia.

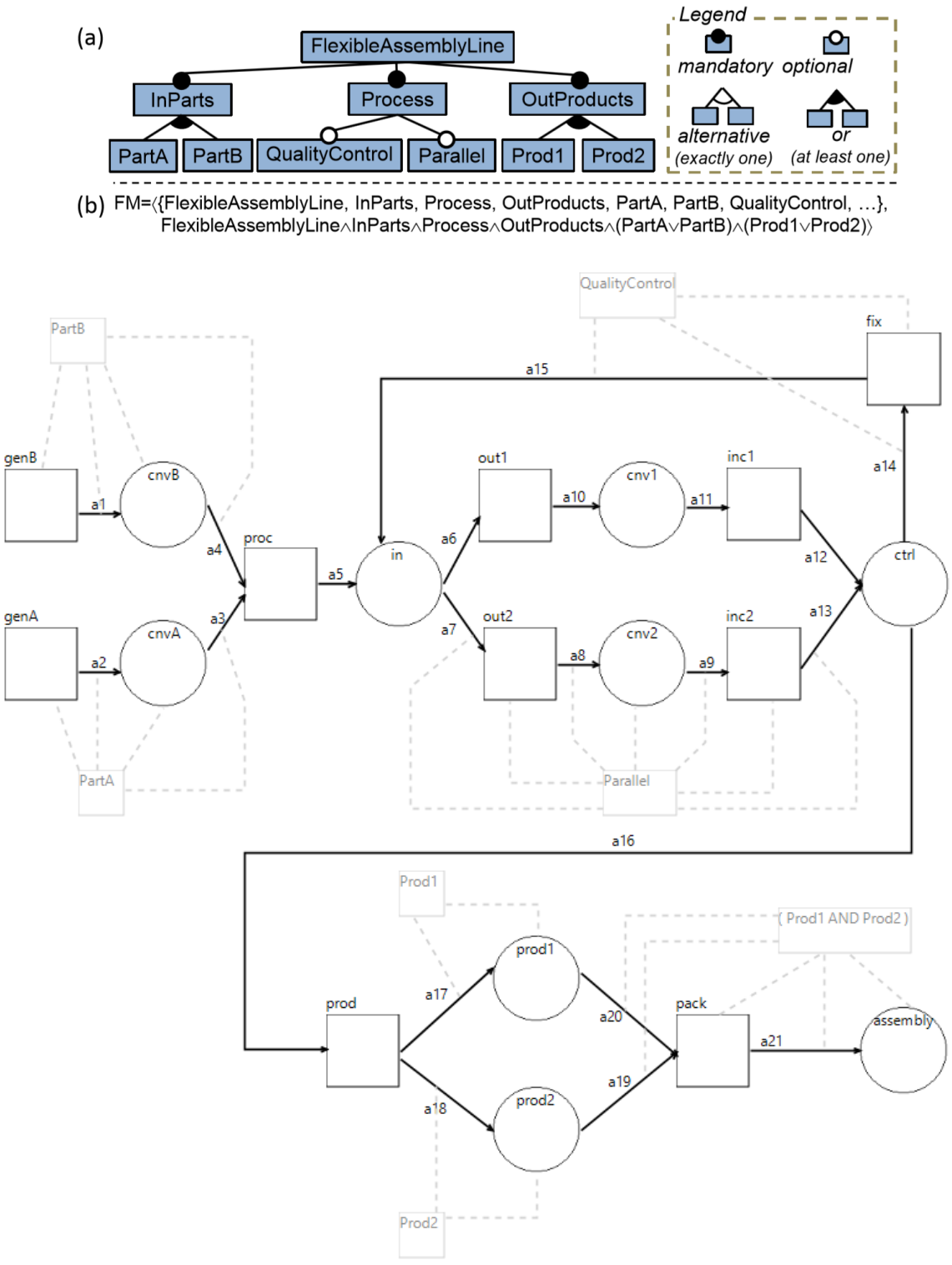

Figura 6-2. Modelado de una línea de productos de redes de Petri [4].

6.2 Plug-in de procesado de archivos.

Al igual que en el caso del diseñador, se hicieron una serie de pruebas donde se probaban las limitaciones de las funciones generadas para el proceso de formación de nuevos archivos.

La parte más crítica del código era la inserción de elementos dentro de la estructura de expresiones. Por ello se hicieron una serie de pruebas de caja blanca y caja negra a las funciones de `setExpression` y `setExpressionTree`. Debido a que las soluciones y el flujo que se debía seguir eran correctos, se solventó fácilmente la funcionalidad planteada.

Además de estas pruebas a funciones se pasaron una serie de archivos los cuales eran de cierta complejidad e intentando abarcar la mayor parte de los posibles casos, y también se consiguió pasar esta prueba correctamente.

7 Conclusiones

7.1 Conclusiones

Este Trabajo de Fin de Grado tenía como objetivo la generación de una herramienta de modelado de Líneas de Producto de redes de Petri. El resultado final ha sido un *plug-in* para Eclipse en el que se podían modelar de forma sencilla PNPL para más tarde ser analizadas por algoritmos.

> A lo largo de todo el proyecto se han tratado una serie de tecnologías a modo de complementos dentro del programa Eclipse que interaccionaban entre ellos, con las cuales no estaba familiarizado y se han tenido que aprender una serie de conceptos para poder utilizarlos, que ha dificultado en gran medida el desarrollo del proyecto.

En los diversos puntos del desarrollo ha habido tramos que han sido complicados y desafiantes debido a que, para entender cómo funcionaban componentes de Sirius que no eran triviales, se ha tenido que hacer una búsqueda exhaustiva por foros ya que los tutoriales explicaban solamente casos sencillos de uso.

Pese a ser el desarrollo de un módulo de eclipse, yo desde mi punto de vista considero que ha sido un trabajo interesante por todo lo aprendido para su desarrollo, desde los conceptos teóricos, hasta su funcionamiento.

7.2 Trabajo futuro

Esta herramienta puede ser ampliada con una serie de trabajos:

- Modificar la forma en la que se han ido generando los *tokens* dentro de los lugares de tal forma que, si hay un número alto de elementos, se exprese mediante un *token* y un número.
- Añadir funcionalidad a las transiciones y buscar la forma de poder hacer que el complemento pueda ejecutarlas. Con esto obtendríamos una red de Petri dinámica, aunque este caso creo que con los materiales y herramientas que se dispone no podría ser un simple complemento, sin una aplicación nueva.

Además de estas mejoras, se plantea la posibilidad de usar los archivos salida del procesador de ficheros como entrada a una aplicación en la que se añada un *feature model* que cumpla las condiciones necesarias para ser acoplado a estas redes de Petri y formar así una línea de producto concreta de redes de Petri.

Referencias

[1] M. Silva, Las Redes de Petri: en la Automática y la Informática, Editorial Ac, 1985.

[2] L. D. Murillo, « Redes de Petri: Modelado e implementación de algoritmos para autómatas programables,» *Tecnología en Marcha*, 2008.

[3] J. G. E. C. M. &. S. R. de Lara, «Model transformation product lines,» de *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018.

[4] E. d. L. J. &. G. E. Gómez-Martínez, «Extensible Structural Analysis of Petri Net Product Lines.».

[5] E. L. J. D. &. G. E. Gómez Martínez, «Towards extensible structural analysis of Petri Net product lines.,» 2019.

[6] «Petri Net World: Online Service for the Internacional Petri Nets Community,» [En línea]. Available: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/. [Último acceso: 27 mayo 2020].

[7] «PNML reference site,» [En línea]. Available: http://www.pnml.org/.

[8] P. L. C. M. P. R. &. K. W. J. Bonet, «PIPE v2. 5: A Petri net tool for performance modelling,» de *Proc. 23rd Latin American Conference on Informatics*, 2007.

[9] «Platform Independent Petri net Editor 2,» [En línea]. Available: http://pipe2.sourceforge.net/. [Último acceso: 22 mayo 2020].

[10] D. d. I. -. U. d. Torino, «Página de inicio de GreatSPN,» [En línea]. Available: http://www.di.unito.it/~greatspn/index.html. [Último acceso: 30 mayo 2020].

[11] M. B. D. C. M. D. P. S. D. &. G. F. S. Baarir, «The GreatSPN Tool: Recent Enhancements.,» *ACM Performance Evaluation Review Special Issue on Tools for Performance Evalaluation*, 2009.

[12] T. &. A. P. Freytag, « WoPeD goes NLP: Conversion between Workflow Nets and Natural Language,» *BPM (Dissertation/Demos/Industry)*, 2018.

[13] «Homepage of ITS-tools | ITS Tools,» [En línea]. Available: https://lip6.github.io/ITSTools-web/. [Último acceso: 17 mayo 2020].

[14] E. Foundation, «Eclipse,» [En línea]. Available: https://www.eclipse.org/. [Último acceso: 18 marzo 2020].

[15] E. Foundation, «Eclipse Modeling Project,» [En línea]. Available: https://www.eclipse.org/modeling/emf/. [Último acceso: 25 marzo 2020].

[16] «Object Constraint Language (OCL) tutorial,» [En línea]. Available: https://modeling-languages.com/ocl-tutorial/. [Último acceso: 22 abril 2020].

[17] E. Foundation, «PDE | The Eclipse Foundation,» [En línea]. Available: https://www.eclipse.org/pde/. [Último acceso: 28 abril 2020].

[18] «Sirius - The easiest way to get your own Modeling Tool,» [En línea]. Available: https://www.eclipse.org/sirius/. [Último acceso: 17 febrero 2020].

[19] «EObject (EMF Javadoc),» [En línea]. Available: https://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/EObject.html. [Último acceso: 12 abril 2020].

[20] E. Foundation, «Sirius/Tutorials/StarterTutorial - Eclipsepedia,» [En línea]. Available: https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial. [Último acceso: 10 mayo 2020].

[21] «Eclipse Sirius Documentation,» [En línea]. Available: https://www.eclipse.org/sirius/doc/. [Último acceso: 10 junio 2020].

[22] R. &. L.-C. M. S. Rodríguez-Puente, «Algoritmo de reducción de grafos sin pérdida de información,» *Computación y Sistemas,* 2014.

Glosario

E

ecore: extension del fichero conde se guardará un meta-modelo. Su nombre hace referencia a núcleo.
EMF: Eclipse Modeling Framework.

F

Feature: Cada una de las caracteristicas en las que difieren los elementos que componen una Linea de Productos de redes de Petri.

G

Geist3D: Extension de Visual Studio para el modelado de redes de Petri.
GreatSPN: Herramienta para el modelado de redes de Petri.

I

ITS Tools: Complemento de Eclipse para generar redes de Petri utilizando código.

L

Lugar: Elemento de una red de Petri que representa una fase donde se acumulan elementos en un estado.

M

Metamodelos: Nombre que se le da a un modelado de clases que se ha obtenido analizando un problema.

O

OCL: Object Constraint Language.

P

PDE: Plug-in Development Environment.
Plug-in: Complemento o extension de un programa con el objetivo de que este tenga nuevas funcionalidades.
PNML: Petri Net Markup Language.
PNPL: Petri Net Product Line.

R

Red de Petri: Formalismo que permite representar un sistema concurrente.

S

Sirius: Extensión de Eclipse con el objetivo de crear herramientas de modelado.

T

tokens: Elemento de una red de Petri que en su conjunto pretenden representar una situacion exacta del sistema.
Transicion: Elemento de una red de Petri cuyo objetivo es aplicar una funcion a los elementos situados en un lugar.

W

WoPeD: Herramienta para el modelado de redes de Petri.
Workflows: Flujo de trabajo.

X

XMI: XML Metadata Interchange.
XML: Extensible Markup Language.

Y

YAWL: Yet Another Workflow Language.

Anexos

A Manual de usuario

Para facilitar un uso sencillo del proyecto generado durante el transcurso de este Trabajo de Fin de Grado, se ha creado este Manual de Usuario. En él daremos unas pautas sencillas para su instalación, el uso básico de las distintas funcionalidades de la extensión, y los posibles errores que se pueden cometer cuando se utiliza.

A.1 Requisitos previos

Antes de poder utilizar la extensión de Eclipse, hay que tener en cuenta el cumplimiento de una serie de requisitos previos.

- La versión de Eclipse debe ser de 2019-06 o superior.
- Tiene que tener instalado previamente los paquetes relacionados con EMF
- Tiene que tener instalado Sirius 6.2.2 o una versión más reciente.

A.2 Instalación de la extensión de eclipse

Como cualquier extensión que se puede añadir a Eclipse, debemos seguir los siguientes pasos:

1- Iniciar Eclipse.
2- Una vez iniciado, en la parte superior de Eclipse, seguimos la ruta *Help->Install New Software.*
3- En este momento se abrirá una ventana como la siguiente:

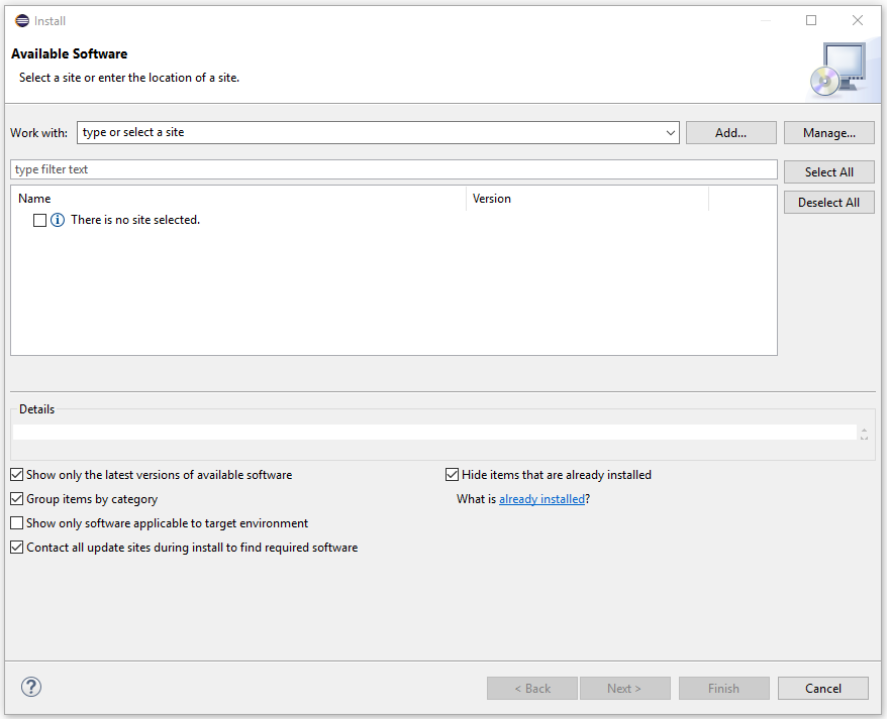

Figura A-1. Pestaña de instalación de nuevo software.

En esta ventana, seleccionamos el botón *Add* en la parte superior derecha.

4- Ahora, aparecerá el siguiente *pop-up*:

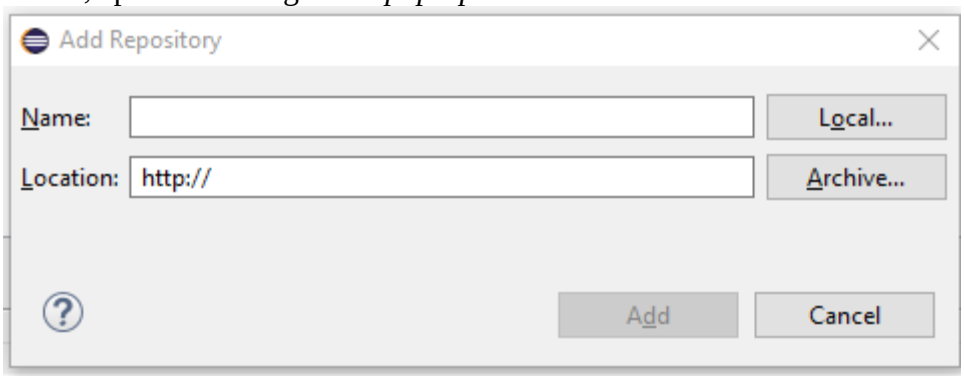

Figura A-2. Pestaña de búsqueda de nuevo Software.

En este caso, al ser nuestra extensión un comprimido de tipo *zip*, deberemos pinchar el botón *Archive*. En este instante se nos abrirá el explorador de archivos de *Windows*, y buscaremos nuestro comprimido. Una vez encontrado se selecciona en *Add* y se cerrará el *pop-up*.

5- En este momento volveremos a la ventana mostrada en la Figura A-1. La única diferencia que tendrá es que habrá elementos en la parte interna situada en el centro, los cuales tendremos que marcar antes de dar *Next.*

6- A partir de este punto con simplemente aceptar las licencias y continuar hasta el final, se debería proceder a la instalación.

7- Una vez finalizada la instalación de la *feature,* Eclipse pedirá un reinicio para activar nuestra extensión.

A.3 Elementos básicos

A.3.1 Herramienta de diseño de redes de Petri

Como bien se ha ido explicando a lo largo de todo el documento, la funcionalidad principal de nuestra *feature,* es una herramienta que sirve para diseñar redes de Petri desde cero. Para poder utilizarla, la manera más sencilla seria la siguiente:

1- En la ventana inicial de Eclipse, seguimos la ruta *File->New->Other.*

2- Aparecerá una ventana cuya funcionalidad es un buscador de tipos de proyectos o archivos:

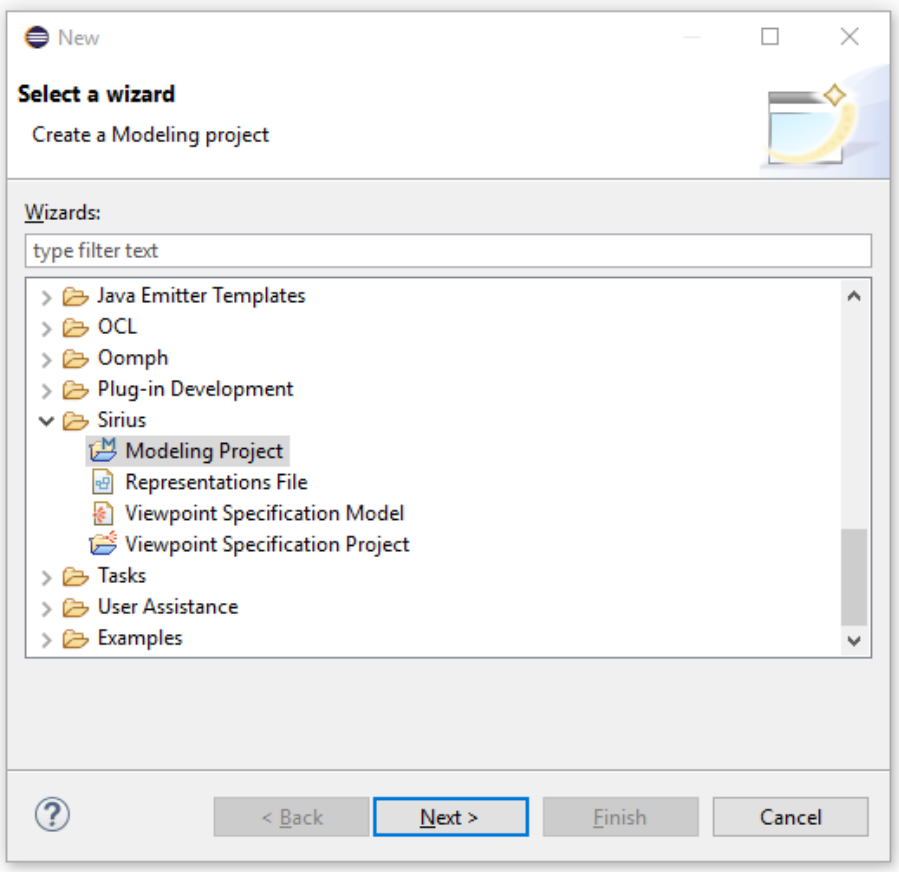

Figura A-3. Ventana de creación de proyecto.

Dentro del buscador, tenemos que buscar la carpeta de Sirius, para poder escoger *Modeling Project.* Una vez escogido, en otra ventana indicaremos el nombre del proyecto y terminaremos su creación.

3- En este instante, y con el proyecto de Sirius creado, crearemos nuestro tipo de archivo. Para ello, volveremos a seguir la ruta *File->New->Other.* En este caso deberemos ir al apartado *Example EMF Creation Wizard,* y escogeremos el tipo *Pnmlcoremodel Model*, lo pondremos un nombre y lo añadiremos al proyecto creado anteriormente.

4- Durante la creación de este archivo nos pedirán un elemento raíz para trabajar el diseño. En nuestro caso y con el diagrama de clases que se utilizó, elegiremos como *Model Object* la clase *PetriNetDoc.*

5- Volviendo al proyecto generado anteriormente, para poder usar nuestra herramienta, deberemos abrir el archivo *representations.aird.*

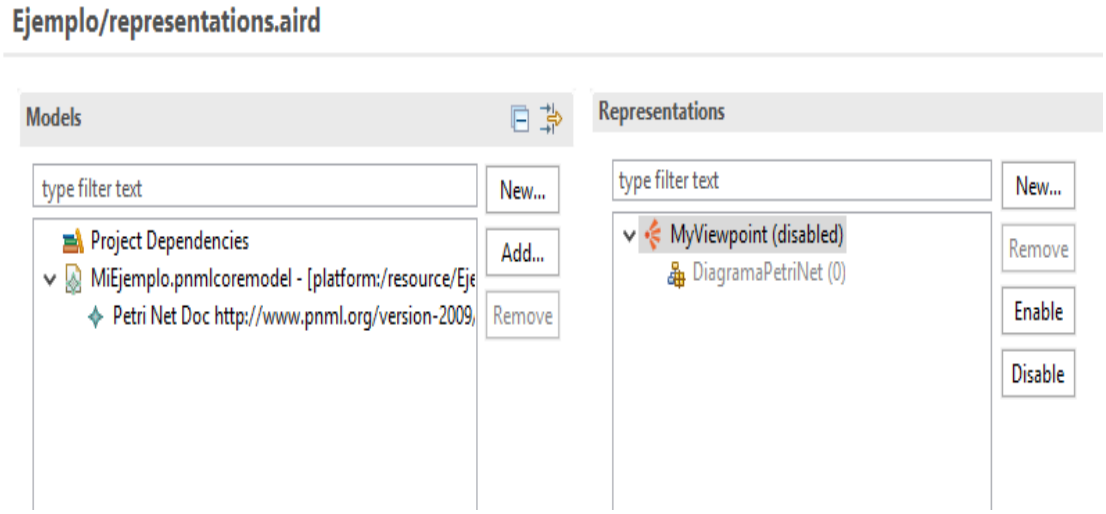

Figura A-4. Vista del archivo representations.aird.

Podemos ver en el editor que este fichero tiene dos apartados, uno llamado *Models* que en nuestro caso lo hemos trabajado en el paso 3, y otro denominado *Representations*. En este último, habilitaremos *MyViewPoint* haciendo doble click.

6- Acto seguido, pincharemos *DiagramaPetriNet* y nos saltará otra ventana donde escogeremos *Petri Net Doc* y finalizaremos, y nos pedirá un nombre para la representación.

En este momento, se abrirá automáticamente la herramienta de creación que se utilizará para representar nuestra Línea de Producto de Red de Petri.

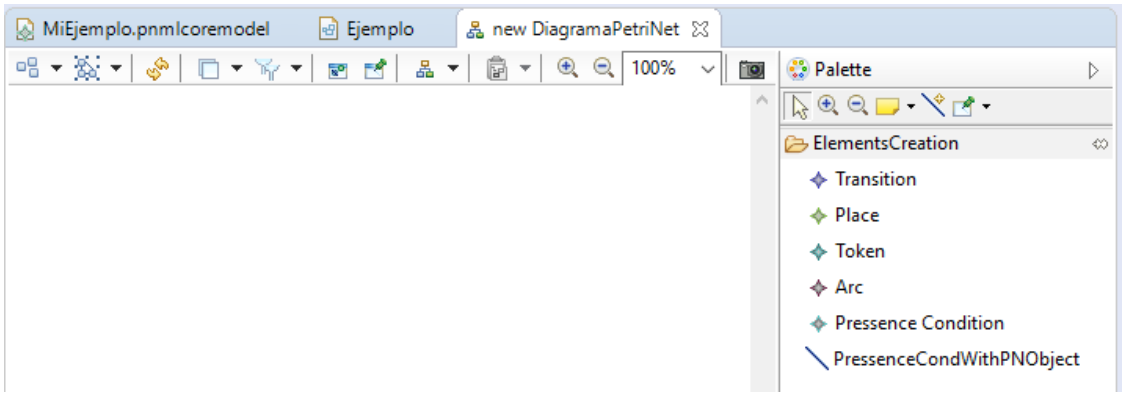

Figura A-5. Vista de la herramienta de diseño.

Los elementos que aparecen a la derecha, en el apartado de *ElementsCreation* son aquellos de los que se puede componer la red de Petri. Estos se pueden dividir en dos grupos atendiendo a la forma de utilizarlos. Estos grupos son los siguientes:

1- Grupo de elementos independientes. Son aquellos que no dependen de otros elementos para su creación en el diagrama. En este grupo estan:

- *Transition.*
- *Place.*
- *Presence Condition*

2- Grupo de elementos que dependen de otros elementos. En este caso, será necesario pinchar encima del elemento independiente que corresponda en cada caso. Las restricciones que tienen estos elementos son las siguientes.
    - *Token*: Se deberá crear en el interior de cualquier *Place* tantos como se quieran.
    - *Arc*: Se utilizará para representar la unión entre *Place* y *Transition.* Para utilizarlo, una vez seleccionado este elemento, se ha de pinchar encima del elemento fuente y después pinchar el elemento destino.
    - *PresenceCondWithPNObject*: Se utilizará para representar la unión entre una *PresenceCondition* junto con cualquier otro elemento representado en el diagrama, a excepción de *token.* Por motivos de implementación la unión se deberá empezar en el elemento *PresenceCondition*, pero los pasos a seguir son iguales que para *Arc.*

Sabiendo cómo funcionan todos los elementos que pueden componer el diagrama de clases de la red de Petri que queramos representar, podemos empezar a trabajar.

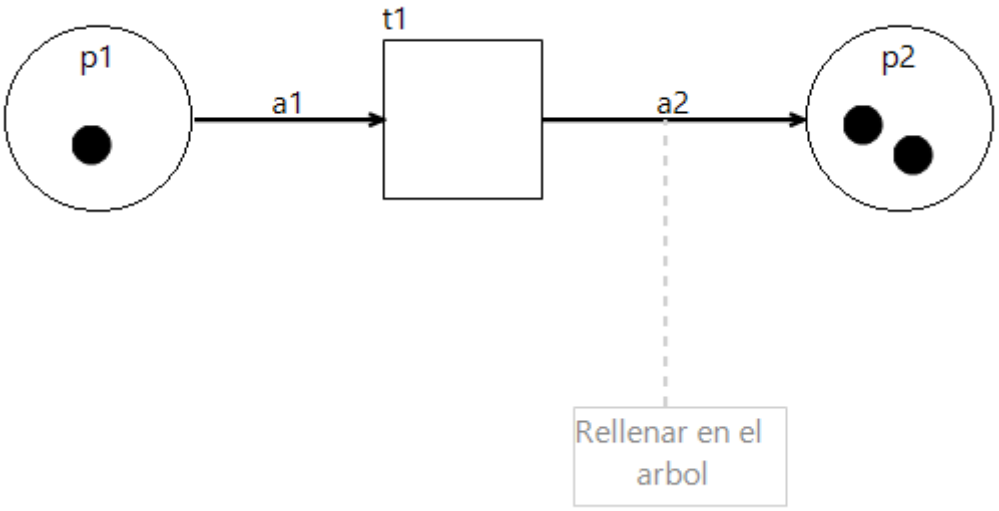

Figura A-6. Red de petri con todos los elementos que la componen.

Como se puede apreciar en la Figura A-6, se han añadido todos los tipos de elementos que pueden componer el diagrama de una forma simple y sencilla. Lo más importante a destacar de esta imagen es que en el elemento *Presence Condition* hay un texto escrito que indica que hay que rellenar el árbol. Con esto se refiere a que dentro del archivo de extensión pnmlcoremodel vayamos a la parte de la variabilidad, a la clase *PresenceCondition* y rellenemos con hijos de la misma forma que si estuviéramos usando EMF lo que correspondería a una expresión lógica, que una vez que guardemos, se colocaría automáticamente en el lugar del texto.

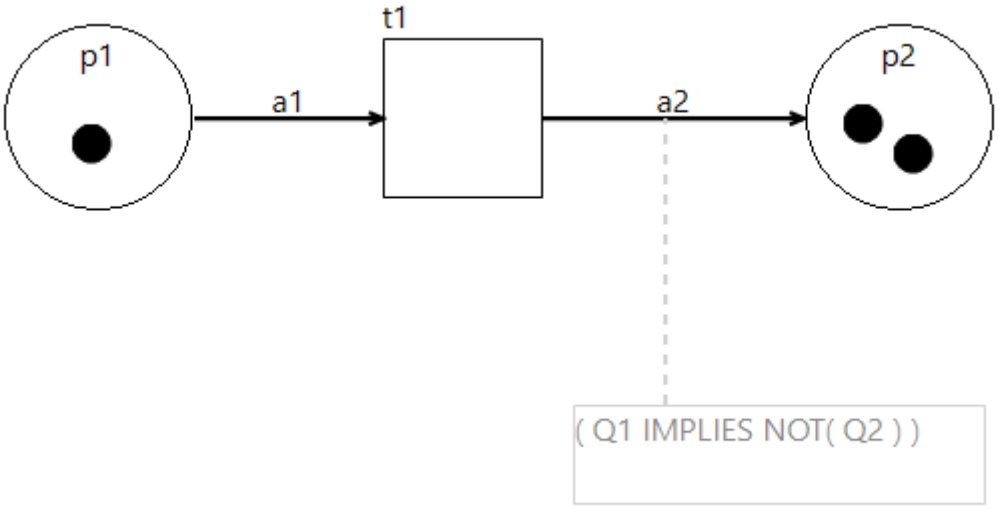

Figura A-7. Adición de una expresión completa al árbol de la *Presence Condition.*

En este momento, ya tendríamos acabado nuestro ejemplo sencillo de red de Petri. Lo último que quedaría por destacar es que se deberían cambiar los valores dentro de la clase variabilidad que hacen referencia a los archivos que utilizamos si fuera necesario.

A.3.2 Complemento de procesado de archivos

Otra herramienta que tiene nuestro comprimido instalable para eclipse es un transformador de redes de Petri generadas en la herramienta de diseño. Su función será, una vez generado y validado un diseño, si el usuario lo desea, la generación de dos archivos en base a la al fichero donde se guarda el diseño:

- Un archivo con extensión .pnmlcoremodel, eliminando la parte de la variabilidad y su aplicación a los distintos objetos dentro del diagrama. Este archivo, será llamado siguiendo la nomenclatura <nombre archivo original>*_cleaned.pnmlcoremodel*.
- Un archivo con extensión .vrb que nos describirá las PresenceCondition que se han aplicado al archivo original en el diseño. Su nombre en este caso será <nombre archivo original>*_annotation.vrb*.

Para utilizarlo, en el menú horizontal superior de eclipse, accederemos a la ruta *Transform Petri Net Variability->Transform*.

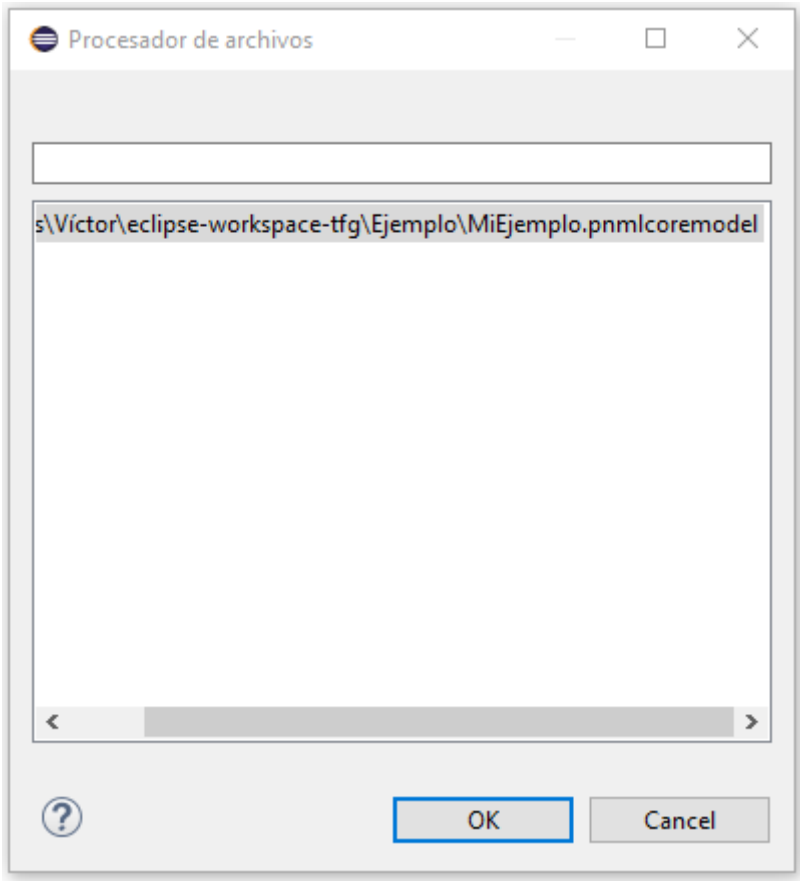

Figura A-8. Selección de archivo del complemento.

Como se puede observar en la Figura A-8, siguiendo la ruta indicada anteriormente, se abrirá esta ventana. En ella se podrá observar una lista que contiene únicamente los ficheros .pnmlcoremodel que haya en el *workspace* donde está trabajando eclipse. En este momento, podremos elegir el archivo del cual queremos generar los dos archivos ya mencionados. Estos archivos aparecerán inmediatamente en la misma carpeta donde se encuentra el .pnmlcoremodel elegido para la transformación. Es importante destacar que, si volvemos a trabajar sobre el original y utilizamos el complemento, los archivos serán sobrescritos.

B Diagramas de clases de los metamodelos

B.1 Diagrama de clases de pnmlcoremodel

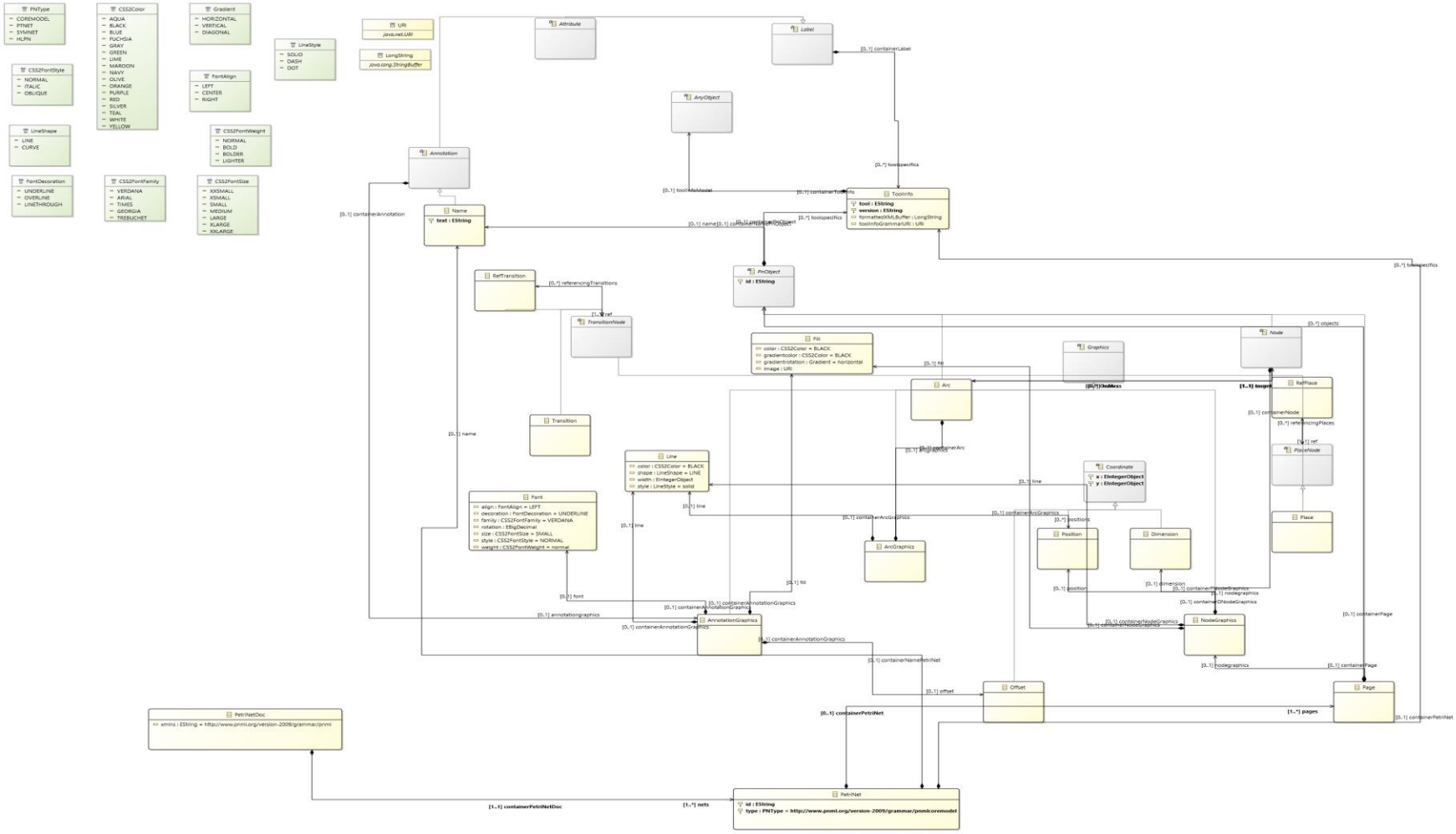

Figura B-1. Diagrama de clases del metamodelo pnmlcoremodel

B.2 Diagrama de clases de variability

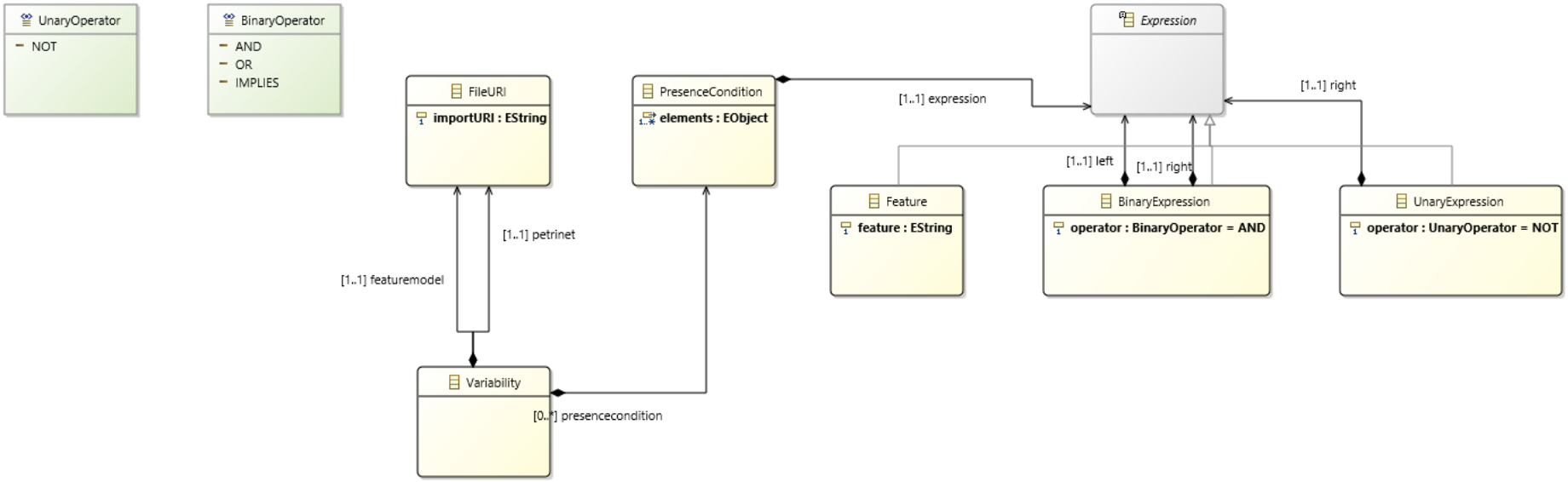

Figura B-2. Diagrama de clases para el metamodelo de variability

C Desde autómatas finitos hasta grafos reducidos

Este anexo, pretendía hacer un recorrido introductorio a los diversos modelos propuestos para solventar el problema de la representación de un sistema concurrente, pero que finalmente se descartó convirtiéndolo en anexo. Destacar antes de empezar, que este recorrido introductorio estaba basado en el libro de Manuel Silva de redes de Petri [1]

Para entender una red de Petri y lo que representa es necesario comprender previamente diversos formalismos que tratan de representar sistemas mediante grafos.

Los autómatas finitos son un modelo de cómputo que representa un sistema cuya única capacidad es pasar de un estado a otro mediante un símbolo acogido en un alfabeto predefinido.

Está compuesto de 5 elementos:

- Un conjunto finito de estados. Éste es importante en marcar que no puede ser de 0 estados, puesto que supondría un problema a la hora de una representación de cualquier máquina de estados.
- Un alfabeto de entrada predefinido.
- Una función de transición que a cada pareja de estados y símbolos le asigna un nuevo estado.
- Un estado inicial donde empezar a trabajar la entrada siempre.
- Un conjunto de estados de finalización en los cuales acabará mínimamente en uno, si la cadena de elementos del alfabeto que entra al autómata es válida.

A estos elementos, se le puede añadir además un sexto opcional que sería una función de lectura o salida, en la cual, cada pareja de estados y símbolos se le asigna un elemento de salida.

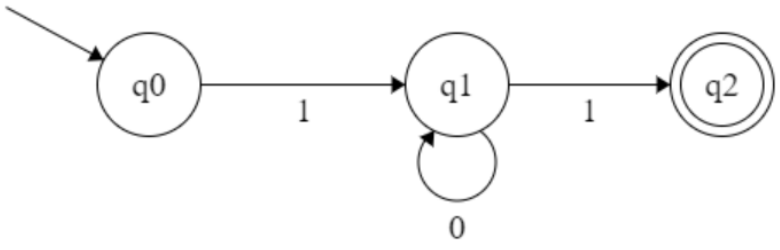

Figura C-9. Autómata que representa una cadena que empieza y acaba en 1 siendo los elementos intermedios 0.

La representación de un autómata finito pretende reflejar todos los movimientos que se harían de un estado a otro mediante las funciones de transición. En la Figura C-9, podemos observar un autómata finito sencillo donde su alfabeto está compuesto por 0 y 1, los estados son q0, q1 y q2 y tiene tres funciones de transición (de q0 con la entrada de 1 pasa a q1, de q1 con la entrada de 0 pasa a q1 y de q1 con la entrada de 1 pasa a q2). Si por ejemplo

quisiéramos aplicar el sexto elemento a la salida, supondríamos que cada transición nos devuelve también un elemento. Por ejemplo, si a q0 se le pasa el símbolo 1 nos devuelve una A. Y así sucesivamente cubriendo todos los tipos de transiciones.

Respecto a este modelo de representación de un sistema, tiene una serie de limitaciones las cuales no nos permiten obtener un número de estados indefinidos y poder trabajar un sistema que necesite trabajar en varios estados de forma concurrente. No obstante, a raíz de este modelo, se pueden obtener otros interesantes cercanos a la idea que estamos tratando de manejar.

Partiendo de la representación de un autómata finito y de su posible tabla de estados, podríamos aplicar el algoritmo de reducción de grafos. Como estos conceptos son introductorios y no requieren un entendimiento matemático, se dirá que el algoritmo de reducción de grafos consiste en reducir el número de elementos de un grafo sin perder información, agrupando los elementos en uno.

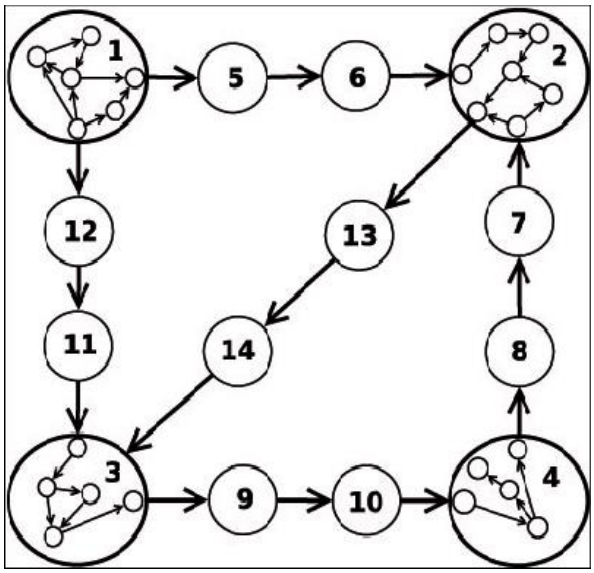

Figura C-10. Grafo reducido [22]

En este punto, sabiendo que buscamos un modelo que pueda representar sistemas concurrentes y de estados infinitos, hemos encontrado algo interesante. Los lugares reducidos son zonas donde se encuentran muchos estados de un mismo sistema. El problema en este caso sería que, conforme más estados, más complejo sería el grafo reducido. Así que para encontrar una solución habría que encontrar otro enfoque al problema. Ese enfoque eran las redes de Petri.